A Mobility-Transparent Model for Consistency

$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$

<http://www.diku.dk/distlab/amigos/pesto.html>

Michael G. Sørensen

{mgsj@diku.dk}

<http://www.diku.dk/students/mgsj/>

Master Thesis

Department of Computer Science
University of Copenhagen
DIKU

<http://www.diku.dk/>

Written Work no. 96-3-7

Not all those who wander are lost;

– J.R.R. Tolkien (Lord of the Rings)

January 26, 2000

Abstract

The design, implementation, and evaluation of a mobility-transparent model for consistency is presented.

A distributed file system with support for mobile computing has been designed and implemented. The system enables applications to utilize any desired level of optimism or pessimism and to adapt their behaviour according to different communication characteristics or user demands. Guidelines for extending the system with a transactional facility are also given.

The distributed file system is based on a model that uses time as a consistency measure. Using such a scheme the applications can relax their consistency requirements as the quality of communication decreases in order to achieve higher availability or reduce cost, and strengthen them again when suited.

The implementation has introduced a small, but acceptable, overhead. The system has some minor flaws, but it is my belief that the implementation has proven the feasibility of the system.

> "The design of a worldwide, fully transparent distributed file system for simultaneous use by millions of mobile and frequently disconnected users is left as an exercise for the reader."
>
> – Andrew S. Tanenbaum (Distributed Operating Systems)

Keywords: mobile computing, distributed file systems, client/server, communication, adaptation, availability, consistency, file sharing semantics, replica control strategies, optimism, pessimism, caching, read and write operations, locks, conflict detection and resolution, transactions.

Preface

This is the Master Thesis of Michael G. Sørensen. It was written as part of the AMIGOS[1] (Advanced Mobile Integration in General Operating Systems) project at DIKU, Department of Computer Science, University of Copenhagen. The subproject was nicknamed $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ for PEssimistic, STrict, and Optimistic.

Prerequisites: The reader is assumed to have knowledge of distributed operating systems corresponding to having read and understood [52, Ch.1-6], e.g., by following the course "Distributed Operating Systems" at DIKU.

The basic ideas of the presented model for consistency stem from the report "Transactions in Mobile Computing" [34], which I recommend is read beforehand, as much of the material in it will be referred to rather than repeated in this report.

A Model for Multi-Level Consistency: During the writing phase of this report, I wrote an article [51] and submitted it for the OOPSLA'96 Workshop on Object Replication and Mobile Computing (ORMC'96). It was accepted and therefore I attended the workshop (in San José, California on the 7. November 1996) and did a presentation. I consider the writing of the article and the presentation as a part of my Master Thesis, and it is therefore the reader will find a copy of the article "attached".

This report (DIKU 96-3-7) is available for download in gzip'ed postscript format: <ftp://ftp.diku.dk/diku/distlab/amigos/diku-96-3-7.ps.gz>. And the attached article as: <ftp://ftp.diku.dk/diku/distlab/ormc96/o12.ps.gz>.

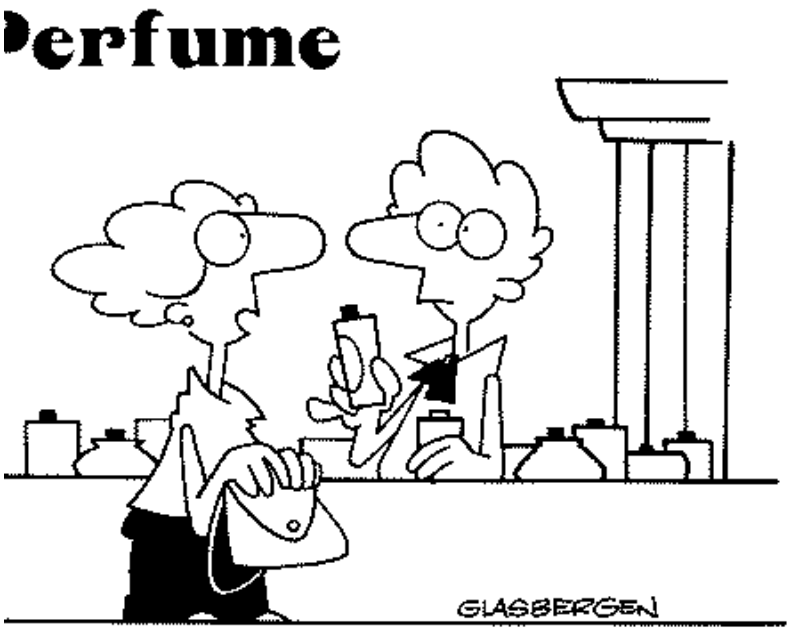
[1] AMIGOS: <http://www.diku.dk/distlab/amigos/>

Acknowledgements: Birger Andersen for pushing me forward, but not over the edge! Jørgen Sværke Hansen & Torben Reich for letting me "steal" a number of items from their TACO Project.[2]

BIG THANKS GO TO

Christina

for her enormous patience



[2]TACO: <http://www.diku.dk/distlab/amigos/taco.html>

Contents

Figures

Tables

Chapter 1

Introduction

This thesis presents the design and implementation of a distributed file service, providing a new set of file operation primitives that enables applications to adapt to the different communication characteristics experienced by mobile computers.

Furthermore, the possibility of extending this new file service with a transactional facility will be discussed.

In the following sections, I present the motivation, the challenges, the goals, and an overview of this thesis.

1.1 Motivation

The advent of mobile computers and the growing popularity of these have inspired many research efforts to integrate and/or support mobile computers within existing distributed systems or even built new ones. The majority of these systems, such as Bayou [53], Coda [22], [42], D-NFS [7], Ficus [13],[1] and Little Work (disconnected operation for AFS) [14], [15], have *a priori* decided to use optimistic replica control strategies to achieve high availability for support of mobile computing, even if conflicting updates are unavoidable and use of stale data possible. On the other hand, (as a bit of an outsider) MIo-NFS [9] uses a pessimistic replica control strategy in order to avoid conflicts, even if this is overly restrictive. Hence, MIo-NFS cannot be expected to have a high level of availability.

[1] Ficus uses optimistic replication in order to scale well (world-wide), but has also found this to be useful in connection with the use of mobile computers.

Since some files (or jobs) are more important than others (to users) and since applications have different file access patterns (i.e., some exhibit higher than average level of write-sharing [28]), why should they be forced to perform perhaps less than optimal due to the shortcomings of either optimistic or pessimistic replication? What if applications themselves could decide on a replication strategy and change it dynamically, e.g., due to change in quality or cost of communication or on demand if a user for some reason wishes to override "default" behaviour? In other words, what if applications could *adapt* their consistency requirements or consistency demands according to the current state of the environment? Adaptation, whether on system or application level (or both), has already been recognized as an important [11] or even essential [43] capability of mobile clients (as members of distributed system).

Transactions can provide mobile computers with an easy-to-use and easy-to-grasp functionality for ensuring correctness or improving consistency, e.g., by detecting read-write inconsistencies [44] (reacting to the use of stale data [16]). Traditional transactions in distributed systems are atomic, consistent, isolated, and durable, but upholding all of these (ACID) properties in mobile environments may not be a viable approach (being overly restrictive [34] or imposing unbearable overhead [40]). A new transactional model that has relaxed some of the four ACID properties is desirable, but in such a way that the functionality maintains its ease of use.

Again, as with replica control strategies, it would make sense to allow applications to adapt to their current environment by providing transactions with different guaranteed levels of consistency.

Thus, the two overall goals are to support mobile computing by:

1. Enabling applications at all times to *adapt* to different communication characteristics.

2. Providing a transactional facility that can be used effectively and efficiently on mobile computers.

1.2 Challenges

This section includes a short summary of the challenges faced when dealing with mobile computers. The AMIGOS project will be presented along with descriptions of two recent works that are of special interest for this thesis.

1.2.1 Mobile Computing

What makes *mobile computing*, that is, the use of mobile computers in connection with distributed systems, so different from traditional distributed computing? A detailed discussion of this issue can be found in Chapter 2, but shortly summarized using the words of [41]:

> "..., there are relative differences between mobile and stationary computers that will always exist and that are not the result of shortcomings of current technology:
>
> - *Mobile computers are resource-poor compared to stationary computers.* Due to constraints on weight, size and power consumption, mobile computers will always be inferior to stationary computers in multiple respects such as processing power and storage capacity. ...
> - *Mobile computers are more prone to loss, damage, and theft than stationary computers.* ...
> - *Mobile computers must operate under a much wider range of networking conditions than stationary ones.* Stationary computers are generally connected to a wired network that has reliable and well-defined characteristics. A mobile one on the other hand must make do with whatever wired or wireless connectivity is available at its current location, which might often be none at all."

These considerations argue for an extension of the traditional client/server model [44]:

> "The relative poverty of mobile elements as well as their lower trust and robustness argues for reliance on static servers. But the need to cope with unreliable and low-performance networks, as well as the need to be sensitive to power consumption argues for self-reliance. ... Any viable approach to mobile computing must strike a balance between these concerns."

One cannot argue with the above observations. They have become well established facts conceived through empirical study of experimental systems such as Coda [42], Ficus [13], and LITTLE WORK [15]. However I find that one important factor, at least, is left unsaid—maybe its to obvious!

- *Mobile computers are normally in use by a single person at a time.* Mobile computers are at current state not powerful enough to support use by multiple users at a time, but even if they were, it would seem unlikely that when out of office (roaming the country) that more than a single user (i.e., the one who carries the computer around) would logon to and use the machine.

I will return to the implications of this simple observation (see also Section 2.3).

1.2.2 AMIGOS

As mentioned in the preface, this thesis is part of the *Advanced Mobile Integration of General Operating Systems* (AMIGOS) project which aims to design and implement extensions for standard operating systems. These extensions should transparently integrate mobile computers with stationary computers, regardless of operating systems and location of the mobile computers [1].

The vision of the AMIGOS project is:

> "... to make the mobile office a reality without making a mobile computer more complicated to use than a stationary computer on a desk."

The AMIGOS project consists of five phases:

- A *transparent communication layer*, which is an extension of TCP/IP with support for disconnected operation, and which—based on availability—transparently selects either an Ethernet, a telephone line, or a cellular phone line.
- A *NFS caching layer*, where the mobile host maintains a large (10Mb-100Mb) cache of files, to enable the user to access files while disconnected or semi-connected.
- A *re-executable transaction scheme* with support for semi-connected operations. This differs from traditional transactions; the mobile host—while disconnected—may be unable to obtain locks for an extended period of time.

- An *object-oriented resource management system*, in which all resources of the distributed system will be made available as objects. These objects may be used for semi-connected operation by extending the transaction scheme to cover these objects.
- A complete *object-oriented environment*, where distributed object-oriented languages like Emerald and Ellie will be extended with support for transactions.

This thesis belongs mainly to the second phase, continuing the investigations of transactions in mobile computing [34] as part of phase three and utilizing the TACO layer [11] developed as part of the first phase. These two works will be presented in the following subsections.

A Mobility-Transparent Model

The model for consistency and availability presented in this thesis stems from the report "Transactions in Mobile Computing" [34]. The model—A Mobility-Transparent Model for Consistency—is based on the use of time as a consistency measure. This is done by associating time bounds with standard file operations (e.g., `read` and `write`). The time bounds tell how high the probability of inconsistency is allowed to be(come).

The basic rules for the model are:

- Weakened consistency should be limited to mobile clients operating weakly connected or disconnected (see Section 2.2.2. & 2.2.3),
  - **Comment**: An immediate consequence of this—since (fully) connected clients should experience no degradation of consistency—is that consistency should, if possible, be restored upon reconnection (i.e., when going from disconnected or weakly connected to fully connected).
- Connectivity and mobility should be concealed from users although mechanisms to discover and influence the weakening of consistency should be provided
  - **Comment**: In my view this boils down to providing applications with ways of *adapting* their consistency requirements to changes in their environment or on demand.

A full description of the (final) model will be given in Chapter 5, so even if seaming a bit abrupt, I will defer from discussing it any further at this point.

Transactions for mobile computing has also been investigated coming to the conclusion that a transactional facility based on the proposed model is not at all an easy task. The main problems arise from the facts that the mobile client may be disconnected at the time when a transaction is executed (on that client) and that consistency cannot be fully guaranteed before the time of commit (on the server).

Traditional transactions are *running* until they are either *aborted* or *committed*. Transactions for mobile computing need to have an intermediate state of *pending*, i.e., they are running or pending (seemingly committed on the client) until they are aborted or committed on the server.

I will look further into these matters in Chapter 4.

Transparent Communication

As part of the AMIGOS project, (in the report "Semi-Connected TCP/IP in a Mobile Computing Environment" [11]) there has recently been developed a transparent communication layer—the TACO (Transparent AMIGOS COmmunication) layer—that hides the details on which communication media to use (and how to use it) from applications.

Furthermore, the work has resulted in support for system and application level adaptation to changes in the environment so that available bandwidth may be used effectively. System level support is supported through means of specifying link requirements[2] of a given type of communication using system calls. These specifications are used by the underlying system (i.e., the communication layer) when scheduling communication that effectively uses available bandwidth. Application level adaptation is provided through system calls, hereby an application can request notification from the communication layer, when changes occur that affect the state of communication.

[2]Link requirements are given through Quality of Service (QoS) parameters.

1.3 A Distributed File System

In the following sections I will define (and confine) what kind of mobile computing system—the *environment* and the *services*—I am working on.

1.3.1 Environment

The environment in this project consists of a single (trusted) stationary file server that services multiple (untrusted) mobile clients. The types of mobile clients supported are discussed in Section 2.1. The service provided is access to shared data (files). The server can be considered as the true home of the shared data and the mobile clients as caching sites as in the traditional client/server model.[3]

In my environment, a mobile computer can be either connected to (the same network as) the server or disconnected. The connections can be by use of a variety of different technologies, such as Ethernet, dial-up telephone line, dial-up wireless communication, see Figure 1.1. Hence, the communication bandwidth may vary substantially as does the cost of using this bandwidth, see Section 2.2.

The server should have no explicit support for stationary workstations, but of course, they can be considered as odd cases of mobile clients that never move and are always (fully) connected, see Section 2.2.1.

The mobile clients are in contact with the server on a regular basis, i.e., they do not stay disconnected forever[4] and can be said to be permanent members of the distributed system in which the server resides. The server does not support "visitors", i.e., mobile clients that do not fall into the category above, and for the sake of simplicity assumes that they do not exist (so that security concerns can be ignored).

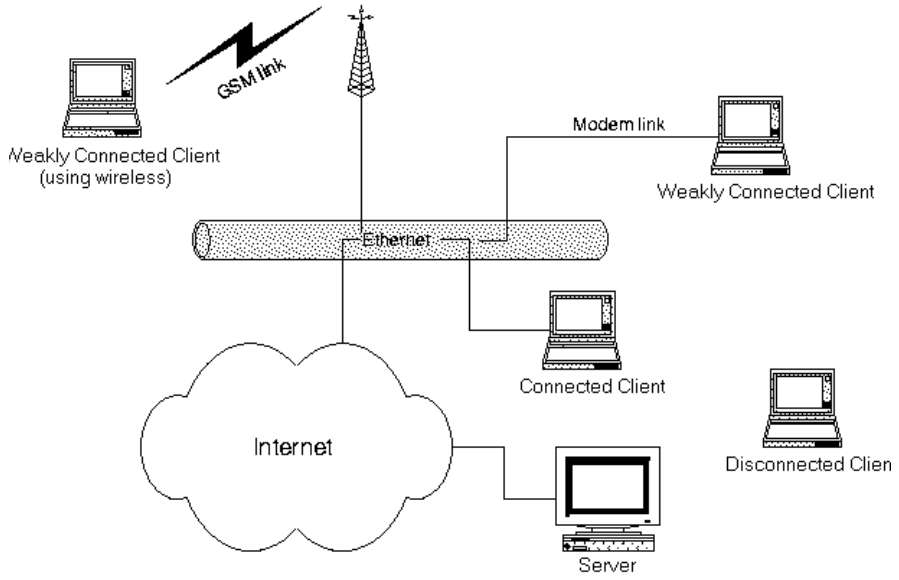
1.3.2 A Distributed File Service

The main work of this thesis is design and implementation of a file system with support for mobile computing. This will be done by providing a new set of file operation primitives that provide applications with access to shared files using any desired level of optimism or pessimism.

[3]The similarities with the traditional client/server model end when the mobile clients are no longer (fully) connected, and varying degrees of self-reliance are needed.

[4]Except in the case where a client for some reason is no longer used (because it is replaced by a newer and better one or simply refuses to work any longer)—but that is a special case that requires special attention anyway!

Figure 1.1: System overview



Precise definitions of design, implementation, and performance goals are given in Section 1.5.

1.4 Transactions in Mobile Computing

The traditional properties of transactions: atomicity, consistency, isolation, and durability (ACID) need to be re-defined, extended, weakened or maybe even abandoned in order to provide an efficient transactional facility for mobile computing.

I will turn my attention towards transactions in (relational) database systems (see, e.g., [4], [36]), since that is where they come from in the first place. I hope to encounter (stumble upon) ideas for a new and improved transactional foundation for mobile computing. Even if some of the ACID properties need to be relaxed or abandoned, transactions should still be an easy-to-use and easy-to-grasp mechanism for achieving higher consistency.

To lay down useful guidelines for a future implementation of an efficient transactional facility on top of the provided file system will be the main objective of the discussion.

1.5 Goals

"Always make more promises,
than you can break"
– D:A:D (Unowned)

In the following subsections specific design, implementation, and performance goals will be given.

1.5.1 Design

The three main goals for the design of the distributed file system are (in order of priority):

1. It should enable applications to *utilize any desired level of optimism or pessimism*,
2. it should enable applications to *adapt their behaviour according to different communication characteristics* (or user demands), and
3. it should be easy to *port existing application to use the new file system.*

I believe that the first goal can be achieved by basing the implementation on the model proposed in [34], and the second goal by utilizing and refining the facilities provided by the TACO layer [11]. The third goal will be achieved by designing a well-defined, easy-to-use, and easy-to-understand (simple) interface to the services provided by the new file system.

By reaching these goals I hope to be able to design a distributed file system that is well suited for mobile computing by being simple, yet flexible and powerful.

1.5.2 Implementation

Communication:
The services will be provided via specialized file operations supported by software[5] that is to be linked with (new) applications wishing to communicate with the server. For the purpose of communication, `sockets` will be used.

Use of TACO:
The system level support for adaptation in TACO enables applications to set Quality of Service (QoS) parameters for each `socket` connection using simple system calls.

[5] A client stub implemented as a `C` Runtime Library

The application level support for adaptation in TACO comes in form of API support for link handling. By choosing an implementation using `sockets` in `C`, it is made possible to use the facilities for adaptation provided by TACO.

The goal of the implementation will be to prove the feasibility and viability of the design. I believe that the environment described in Section 1.3.1 and the choice of using `sockets` for communication purposes provides an adequate setting for achieving this goal.

1.5.3 Performance

The new set of file operation primitives will require additional administration (especially on the client side) compared to the corresponding "ordinary" file operation primitives. The overhead introduced by this administration should be negligible.

Furthermore, performance degradation should only occur when unavoidable, e.g., due to weakened connectivity. In other words, fully connected clients should not be punished; they should not experience any non-negligible degradation of performance.

1.5.4 Evaluation

I will evaluate the system's fully connected operation by performing time measurements on operations performed by the same fully connected notebook computer using only the TACO layer and using the new file system (on top of TACO) and comparing these. If my performance goals are to be fulfilled, the result should show no more overhead than can be explained by the implementation of the server and client software (using the `socket` facilities) on top of the existing system with TACO.

The system's weakly connected operation will be evaluated by performing the same set of operations as above with a known and stable amount of available bandwidth. The running times should be comparable to those of fully connected operation when the lower bandwidth (and the higher latency) is taken into account.

1.5.5 Restrictions

"Where there's a will;
there's a won't..."
– D:A:D (Unowned)

I will restrict my self from migration issues. The mobile clients will (must) contact the server, not the other way around, i.e., I will ignore the fact that mobile computers may wish to migrate between different networks. Readers interested in this are referred to [11]. I will not take any considerations with regards to security (e.g., authentication and data encryption).

This thesis is a continuation of the work presented in [34]. To avoid unnecessary repetition of some of the material, I will skip in-depth treatment of the following issues:

- Serializability-based synchronization (e.g., Two-phase locking, Optimistic concurrency control, and Timestamps).
- Atomicity and recoverability (e.g., Stable storage, Logging, Shadowing, and Two-phase commit).
- General description of distributed file systems (e.g., Network File System, Andrew File System, Little Work, and Coda).
- The role of a mobile host [in transaction processing] (e.g., Remote terminal, Full-fledged server, and Preprocessing client).

For my own sake, and because my views differ slightly from those given in [34], I will be repeating the parts about mobile-computing in general and basic properties of transactions. Furthermore, I find it natural to include a full description of the implemented model for consistency.

1.6 Overview

1.6.1 Terminology

Throughout the rest of this report I will use the term *mobile computer* to mean portable computers in general, and the term *mobile client* to mean a mobile computer used in connection with a distributed system, i.e., a mobile computer that communicates (regularly) with one or more servers. In the same way the terms *stationary computer* and *stationary client* are used. Servers are assumed to be stationary.

1.6.2 Contents

The thesis is structured as follows:

- Chapter 1 contains the Introduction which I will assume you have just read; otherwise do it!
- Chapter 2 is about Mobile Computing; the types of (mobile) computers supported will be determined and the characteristics of mobile computing given.
- Chapter 3 is a general discussion of the issues concerning replica control in connection with caching of files. Issues touched upon include file access patterns, granularity, transparency, replica control strategies (pessimistic, strict, and optimistic), synchronization, and file caching strategies.
- Chapter 4 is about Transactions. Basic properties of transactions and concurrency control will be discussed.
- Chapter 5 contains a full description of The Model, i.e., a file service specification for the system.
- Chapter 6 contains Implementation issues.
- Chapter 7 gives a description of the tests performed in order to evaluate the system with respects to usability and performance, and of course, a discussion of what the results of these tests tell us is included.
- Chapter 8 is the Conclusion. As with all thesises the conclusion comes last telling how well it all went. References to related work and ideas for future work will also be given.

Chapter 2 to Chapter 4 each finishes with a summary.

The interested reader can find a listing of excerpts from the source code in Appendix A and detailed descriptions of the examples used within the report in Appendix B. Appendix C contains flow diagrams for selected parts of the client software, and finally Appendix D contains Figures 5.4, 5.5, and 5.6 from Chapter 5 in full sizes for better viewing.

Chapter 2

Mobile Computing

The term *mobile computing* is used in connection with a wide range of different computer environments (and other systems), see, e.g., [24].

In this chapter I will determine the type of *computers* supported. Furthermore, characteristics of mobile computing will be given with emphasis on special considerations to be taken when dealing with mobile computers, i.e., where they differ (significantly) from "stationary" computers.

2.1 Mobile Computers

CELCUS ULTRA POSSE NEMO OBLIGATUR

In my environment mobile computers should be *self-contained*, i.e., they should be fully functional computers with their own operating system (e.g., Windows95, OS/2, or Linux) and applications—allowing the user to work independently from other machines (e.g., servers). In my particular case the operating system chosen is Linux, since it is currently in use on the mobile computers used in connection with the AMIGOS project, and it has `sockets`. In the long run any operating system providing a `socket` abstraction (in `C`),[1] so in this sense the mobile computers supported are *heterogeneous*.

In Bayou [53] the mobile computers must run some sort of Posix compliant operating system (actually, it could run in the same setting as mine) which confines the use to some sort of UNIX-clone.

In Coda [22], [42] the mobile clients are not allowed to be heterogeneous, in fact, they do not even come with their own operating system, thus making

[1] It could, I have been told, equally well have been Windows and `winsockets`, i.e., the `socket` facility provided with the former, but I found it best (at first, anyway) to stick with the existing systems.

it necessary to *cache* system files on clients in order for them to work during periods of disconnection from the server(s). Personally, I would find it a bit absurd (and terribly annoying), if the machine I was using, stopped working for the lack of some obscure system file that I have never heard of. This problem can be tricky to handle[2] and even nearly impossible [25].

The people behind the SEER [26] predictive caching system (an extension to Ficus) also caches system files (i.e., the mobile clients are not heterogeneous), and they have found that "interrelationships among programs are complex and deliberately hidden", and therefore users cannot be expected to provide accurate specifications of critical files to cache.

I expect that the need to track down file usage as in Coda (by use of a help utility) or as with SEER (by constantly logging file references) is avoided altogether in my system because all (necessary) system files will be available on the mobile clients as they are self-contained, i.e., come with their own operating systems.

As another part of the AMIGOS project a system MIo-NFS [9] (Mobile Integration of NFS) that integrates mobile computing with NFS has been developed. That system is also heterogeneous in the sense that the clients only have to provide NFS—and nowadays you can even get NFS for PCs [49].

Everybody can agree on the fact that a mobile computer should be *small*, *light* and at least to some extent *handy*. Making them that way necessarily means that they become inferior to stationary workstations or *desktops* in terms of processing power (CPU & RAM), storage capacity (harddisk), and user interface (screen & keyboard). These factors as well as the question of limited power supply will be discussed in the following subsections.

2.1.1 Performance

The vision of the AMIGOS project is [1]:

> "... to make the mobile office a reality without making a mobile computer more complicated to use than a stationary computer on a desk."

This could also be stated as [24]:

> "... mobile computers should be a substitute for the stationary computer at their office."

[2] In Coda [42] they have even developed a special `spy` program to track down use of files during a session.

In terms of computing power this implicates that as a minimum the mobile computers should be comparable to the desktops which (in my view) rules out *PDA*s (*Personal Digital Assistants*) and most *palmtops*, and leaves me with *notebooks* (or *laptops*). In other words, I only wish to support the types of mobile computers that can be roughly characterized as *portable workstations* in [41]. I find the notebooks you can buy today to be adequate,[3] see Table 2.1.

Table 2.1: Memory/CPU

Harddisk	Memory	Machine (CPU)
>1 Gb	100 Mb	Alpha or vector proc.
1 Gb	32 Mb	High-end Pentium Desktop
1 Gb	16 Mb	High-end Pentium Notebook
500 Mb	16 Mb	High-end i486 Notebook (or desktop)
250 Mb	8 Mb	Low-end i486 Notebook (or desktop)
		High-end i386
150 Mb	8 Mb	High-end i486 Subnotebook
60 Mb	4 Mb	Low-end i386 Subnotebook
20 Mb	2 Mb	i186 Subnotebook
-	2 Mb	i186 Palmtop (or PDA)

(adapted from [2])

2.1.2 Stable Storage

At DIKU users (such as myself) normally have a home directory *quota* of 8Mb. Some privileged users have been granted additional 20Mb of disk space, making it a total 28Mb. Space for additional file information and space for file copies and/or logs are also required. A preliminary guess would be that a 100Mb cache size would suffice for a disconnected (see 2.2.3) day's work for (normal) users, e.g., students at DIKU.

Coda has reported that in their environment a cache size of 25Mb for one day of disconnected work or a cache size of 100Mb for a full week of disconnected work should suffice [42]. It must be remembered that they also cache system files requiring additional space compared to my environment. On the other hand, system files are not modified (at least they should not be) during disconnection, so they do not require additional space for file copies or

[3] In terms of computing power; the question of *value for money* is always debatable!

logs. These facts suggest that the preliminary guess of 100 Mb is not totally off.

Nowadays, an additional 100 Mb of harddisk does not cost a fortune, in fact, harddisks with less than 500 Mb is seldom seen anymore.

The operating system and the usual applications that come with it also requires some space (remember that the mobile computer should be self-contained), but how much is entirely up to the user. For Linux it is somewhere between 100 Mb and 1 Gb depending on how much you need/want. Anyway, I will not consider it a concern of mine; I am only concerned with the available space for caching.

2.1.3 Vulnerability

Due to their portability (being carried around and everything) mobile computers are more likely to be stolen and more vulnerable to loss or damage than stationary computers [45]. With this in mind I hope it is clear why I have already indicated that the mobile clients are "untrusted", and why I have chosen the server to be the true home of the shared data.

2.1.4 Screen & Keyboard

Modern notebooks that will cost you less than 20.000,- dkr. (except if you buy an IBM) come with 10-11" colour screens with VGA graphics (640x480 pixels). Though slighty less than the 14-15" colour screens of a standard PCs and somewhat less than the 19" (or more) screens of the X-Window workstations at DIKU, I personally find it acceptable for any of the standard GUI-based window systems (e.g., Windows95, X-Windows for Linux, or OS/2's Workplace Shell).

I find the keyboards (with or without build-in trackball, trackpoint or external mouse) of today's modern notebooks less acceptable than the screens, but I guess that is a "price you have to pay" if you want something portable.

2.1.5 Power Supply

Notebooks rely on finite energy resources, namely the batteries. With this in mind, any additional client processes introduced to enhance disconnected operation should not be too demanding on the system. The main questions are how far is there between power supplies and how much work will be done when traveling from one to the other? Further investigation on this subject

is beyond the scope of this report. I will leave that question to the users and the applications.

2.2 Means of Communication

"Never underestimate the bandwidth
of a station wagon full of tapes."
– Dr. Warren Jackson, Director, UTCS

The stationary workstations in the office are usually connected to the servers through the LAN. The network is wire-based offering continuous, high bandwidth communication. Mobile computers open other connections through the telephone lines (wire-based dial-up using a modem) or using portable phones (cellular dial-up using GSM). Examples of the latency and bandwidth of the different networks are given in Table 2.2.

The telephone lines have lower bandwidth than the wire-based and you normally have to pay some non-negligible fee [34] for the connection and/or the communication time, but connections can be made from almost anywhere (as long as there is a telephone wire to the house).

Wire-less communications, such as GSM, have even lower bandwidth than telephone lines; they are usually more expensive to use, and are less reliable, but in theory they can be used from anywhere (you do not even have to go to the nearest phone (outlet)).

Table 2.2: Networks

Latency	Bandwidth	Network
0.5 msec.	600 Mbps	ATM + FDDI (LAN)
	150 Mbps	ATM (LAN + country)
	100 Mbps	Fast Ethernet (LAN)
1 msec.	10 Mbps	Ethernet (LAN)
	2 Mbps	Wireless Ethernet (LAN)
100 msec.		Internet (interstate)
	100 Kbps	Serial line and V34 modem (country)
500 msec.	10 Kbps	Radio based (GSM)

(adapted from [2])

In the following subsections I will clarify my use of the terms (*fully*) *connected*, *weakly connected*, and *disconnected*.

2.2.1 Connected

Definition 2.1 *A computer is said to be (operating)* fully connected *or merely* connected *with respect to a certain server when it is physically connected to a wire-based network offering continuous, low latency, and high bandwidth communication with the server.*

This is normally the case with stationary workstations communicating with servers through a LAN (e.g., an Ethernet or a Token Ring). These connections are often provided through non-profitable organization—such as schools—and are as such free of charge. Mobile computers that are connected should experience the same level of network utilization (throughput).

The term *strongly connected* [33] is also used—as opposed to *weakly connected*, see Section 2.2.2

Under these (ideal) network conditions it is feasible always to work on the newest version of a file shipping updates rapidly over the network (when needed). It is also feasible to avoid write/write conflicts, i.e., two simultaneous updates done locally (at two different locations, e.g., caches). This can be done either by implementing UNIX file sharing semantics (all operations are enforced a global time ordering and "reads" always return the most recent value [52]) or by keeping track of who is doing what, e.g., using session semantics and a refresh (checking to see if the cached version has not been updated since last time) or a call-back mechanism (notifying holders of cached copies if an update occurs). The possibilities are many (it has an ongoing research field of its own), but it should be emphasized that this (in some flavour or another) is what people—users of distributed (file) systems—are accustomed to, i.e., write/write conflicts do not (or at least very seldom) occur, consistency **and** availability is high.

2.2.2 Weakly Connected

"GET AWAY from it all
but don't *distance* yourself"
– ZENITH notebook advertisement
(Computer World, No.19 (1996), p.23)

Definition 2.2 *A computer is said to be (operating)* weakly connected *with respect to a a certain server when it is able to communicate with the server but is* ***not*** *fully connected.*

This definition covers a wide range of connections; wire-less ones such as radio links (e.g., GSM) or infrared, and connections over a telephone network (e.g., using a modem). The quality of communication can be anything from acceptable to downright miserable. Communication is often characterized as one with high latency, low and varying bandwidth. Furthermore, it is expensive, due to the fact that use of these communication means often is charged.

The terms *partially connected* [18], [19] and *semi-connected* [10] are often used meaning the same.

Under these (often far from ideal) conditions consistency may have to be weakened in order to increase availability—the communication is too slow, unstable, and/or expensive to use for keeping the files up-to-date, i.e., consistent with one another or even a single primary copy (as in my case).

Still, some communication may be preferred to none, since it gives the possibility of validating the status of cached files—using short messages between the client and the server—and/or receiving or sending files in cases where it is of great importance to have the newest versions or to make updates visible to others as fast as possible.

2.2.3 Disconnected

Definition 2.3 *A computer is said to be (operating)* disconnected *with respect to a certain server when it is unable to communicate with the server.*

This will be the case when there is no telephone service available and the server is out-of-reach through any wire-less means of communication. In this case self-reliance and self-dependence are necessities.

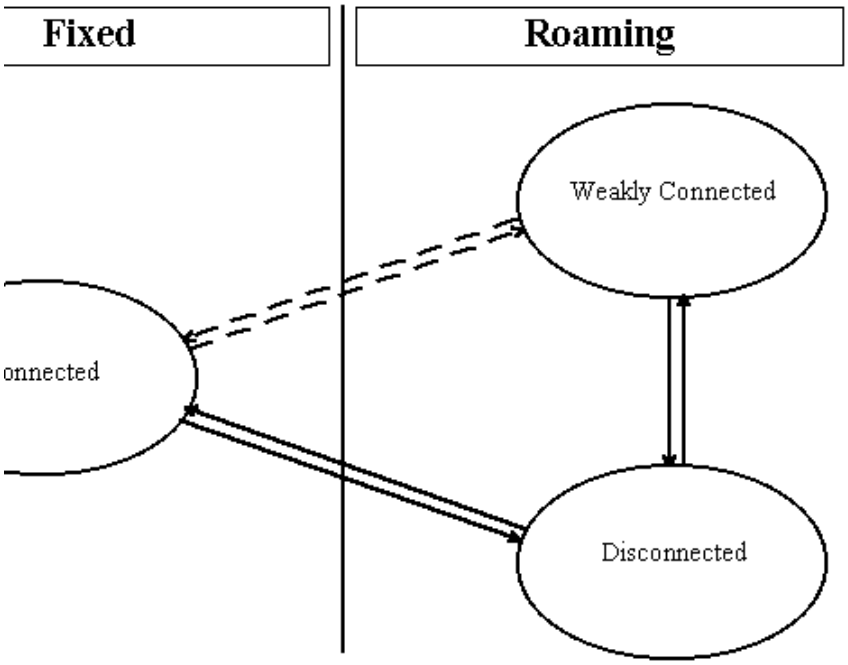
It has been suggested [38] that in the future it will become unnecessary to discuss disconnected operation and that some sort of connection will always be and you should discuss different levels of connectivity and how to switch between *domains*. On the other hand [54] disconnected operation will undoubtedly always be cheaper (in terms of money, since it is free) than any level of connectivity and thus be preferable to the more "expensive" weak connections for (many) years to come—at least as long as the charges are non-negligible.

2.2.4 Communication State Transitions

Well, if the bandwidth is so much greater when connected than otherwise, why do we even bother? Well, we want mobility and as they put in [24]:

> "... with an increasing degree of mobility, the bandwidth of a connection, and as a consequence the amount of data which can be transferred in adequate time, decrease."

Figure 2.1: Different states



Furthermore, disconnected operation may be the result of a network or communication media failure that makes it impossible for the client to stay in contact with the server—an unvoluntary disconnection. The transitions between communication states are shown in Figure 2.1. The transitions between connected and weakly connected are not so common; normally the user of the mobile computer makes a voluntary disconnection before traveling from one place (domain) to another, and then reconnects on arrival. The use of mobile computers is discussed in Section 2.3. As mentioned earlier,

(see page 5) the mobile clients should experience no degradation of consistency when fully connected, and thus consistency should be restored upon reentering connected mode of operation.

2.3 Mobility

A man's computer is a man's computer
mobility is but an illusion
– rewriting of an old Chinese proverb (about faith and life)

I will use this section to pin out what I believe to be the normal use of mobile computers, i.e., who uses them, who wishes to use them, how and when do they use them, and how and when do they wish to use them?

It is my conviction that mobile computers are personal belongings, i.e., they are owned and used by a single person. Today it is not that common to own a mobile computer, but I believe that in a not too distant future this will change and a lot of people will own a mobile computer (on the same scale as people own a PC today). When you buy a mobile computer you know that it is portable and that you might have to use it differently than a stationary computer.

Users of mobile computers know that special attention must be given to situations where they connect to or disconnect from a stationary system and they are willing to do the extra effort it is to follow instructions when doing so. Otherwise they would like to be able to "carry on as usual", i.e., to do the same things on their notebooks, whether they are at the office (and preferably fully connected), on the road (being disconnected or weakly connected), or at home. It should be possible to use the mobile computer in this way without regards to location. The only deviation from this is when the cost is too high, i.e., when it is too expensive to contact the server to get "that missing file".

In Coda [42] they have a hoarding profile (a prioritized list of files to cache during connected operation) pr. client. However that only makes sense if the client is used (or is going to be used) by a *single* person. Otherwise the hoarding profile should be a combination of preferences from multiple individuals—I think not! So they make the same assumptions as I do, only they forget to mention it; maybe it is too obvious.

In Bayou they store a full database on the clients, and that would enable the client to support multiple users, but the developers [54] have plans for partial replication (due to the limited size of client caches, and the potentially growing size of the database) which means that they to must make assumptions of the same sort.

2.4 Summary

I will summarize this chapter by means of three tables that "say it all".

Table 2.3 shows the difference in hardware for the different types of computers; stationary ones (servers & workstations) and mobile ones (laptops & palmtops). From the table it is obvious that processing power and storage capacity (and to some extent also quality of user interface and reliability) is traded for portability!

Table 2.3: Characteristics of computer hardware

	Hardware			
	Stationary		Mobile	
Characteristic	Server	Workstation	Laptop	Palmtop
Processing power	Maximum	High	Medium	Low
Storage capacity	Maximum	High (*)	Medium	Low
Portability	None	Limited	Slightly limited	Full
User interface	-	Full	Slightly limited	Limited
Reliability	High	Medium	Limited	Limited

(adapted from [34])

(*): The stationary workstation itself can be diskless, but then it has access to storage on or through a file server.

Table 2.4 shows the difference in quality of the different network technologies; fixed vs. dial-up wired or wire-less. Fixed is preferable to dial-up, dial-up wired (e.g., using telephone lines) is preferable to dial-up wireless (e.g., using GSM). In both cases the only gain is availability (i.e., mobility).[4]

Table 2.5 sums up the characteristics of the different modes of operation (or communication states, if you prefer) encountered by mobile computers. If I have not expressed myself clearly, this is what I have been trying to communicate. I would like to emphasize that *consistency is weakened* (delayed writes, optimistic replication, and cache reliance) *in order to increase availability*!

As a last remark I would like to point out once again that *mobile computers normally are used by a single person* (at a time).

[4]Also initial cost, but I am talking about servicing mobile clients from an existing distributed system—and hopefully I do not have to worry about financing it.

Table 2.4: Characteristics of network technology

	Network technology		
	Fixed LAN/WAN	Dial-up	
		Wire	Cellular
Operation	Connected	Weakly connected	Weakly connected
Bandwidth	High	Medium	Low
Reliability	High	Medium	Low
Initial cost	High	Low	Low
Latency	Low	Medium	High
Cost to use	Low	Medium	High
Topology	Fixed, continuous	Fixed, varying	Dynamic
Available at	Office outlet	Phone outlet	"Anywhere"

(adapted from [34])

Table 2.5: Modes of operation (states) and their characteristics

	Connected	Weakly Connected	Disconnected
Mobility	none	medium or high (*)	unlimited
Position	fixed	roaming	
Method	normal operation	delayed writes & optimistic replication	cache reliance
Access	continuous	continuous or on demand	none
Bandwidth	high bandwidth constant	low bandwidth varying	none
Latency	low latency	high latency	none
Link	hard-wired e.g., LAN	serial link e.g., phone line or wire-less	none
Network (**)	Ethernet	PSTN or GSM	none
Guaranteed consistency	normal	weakened	none

(*): In theory, high when using wireless; but in practice? When using a phone line medium; you can find a phone outlet in almost every house but you cannot move around during the connection.
(**): Only the network types considered in this project is listed, other possibilities exist, e.g., ATM, WaveLAN, Infrared, etc.

Chapter 3

Replica Control

In this chapter replica control issues, i.e., the matter of keeping replicas of a file mutually consistent, will be dealt with in a (semi-)general manner. The concerns that are specific to the actual model for consistency are dealt with in Chapter 5.

3.1 File Usage

3.1.1 Classification of Files

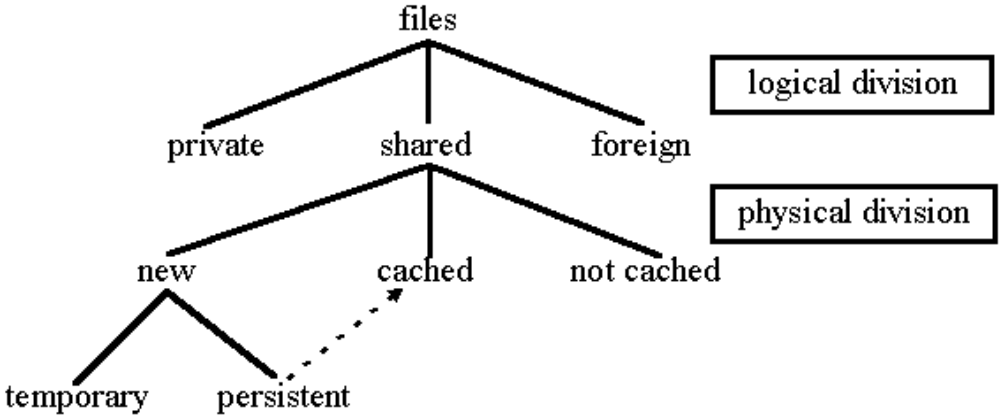
Which files are of concern to a distributed file system when the clients come with their own operating system and applications, i.e., are self-contained? To answer this question I will divide files into three main categories: *private*, *shared*, and *foreign* files. The private files are located on the client and are used only by that client, e.g., operating system files. The foreign files are located on the server or on other clients, but have no relevance to the client, i.e., they are private to the other clients or the server. The shared files are located on the server as primary copies and on the clients, in their caches, as secondary copies. The shared files are those that a distributed file system with self-contained clients should provide access to. Private and foreign files are of no concern to the file system, except for size considerations:

- On the client:
  cache size = storage size - size of private files.
- On the server (as seen from the client):
  size of shared files <= storage size - size of foreign files.

When seen from the client's point of view, a shared file (some version of it) can be either *cached* or *not cached*. If a client creates a new file that is to

be shared then it is (at first) only located in that client's cache, i.e., this is a special case where there is no primary copy of the file (yet)! I will refer to such a file as a *new* file. The classification of files can be seen in Figure 3.1.

Figure 3.1: Classification of files



A new file that is deleted before any version of it reaches the server can be considered a *temporary* file. Contraversely, new files that reaches the server, become cached, and are termed *persistent* when it is necessary to distinguish them from temporary files.[1]

Applications are classified according to which files they use:

- *Private* applications use private files **only**.
- *Foreign* applications use foreign files **only**.
- *Shared* applications use at least one shared persistent file.

An application that uses temporary files but no persistent files should be rewritten to be a private application. The file system should support shared applications; the other applications should be supported by the operating system and the ordinary (local) file system.

[1]Of course, private and foreign files can be divided into temporary or persistent files, but since they are of no concern to the file system, there is no need to make the distinctions.

3.1.2 Operations on Files

Operations on files:

- Reading, writing, creating, and deleting: These are the basic operations on files. In ANSI-C [21] a file is created implicitly when a non-existing file is opened for reading. (Notice that creating and deleting files are also operations on directories,see Section 3.1.3).
- Scanning (positioning): This can be thought of as reading a file until a certain position is reached, without caring about what was read "on the way there".
- Opening and closing: In many systems, and especially those using session semantics it is impossible to read from or write to a file without having opened it (for reading or writing) first. Reads and writes to a file may be performed locally (i.e., without the rest of the system being able to notice it) before the file is closed.
- Linking and unlinking: UNIX synonyms for creating and deleting. The files are not physically removed, but they are linked to or unlinked from the directory structure, and cannot be referenced any more. The space they previously occupied can be garbage collected. (Linking and unlinking files are operations on directories, see Section 3.1.3).
- Updating: Synonym for reading and writing a file.
- Rewriting: A composite operation; rewrite <file> = if <file> exists then delete <file>, create <file>, write <file>.
- Copying: A composite operation on two files; copy <file1> <file2> = read <file1>, create <file1>, write <file2>.
- Renaming and moving: Composite operations on two (or more) files; rename/move <file1> <file2> = read <file1>, create <file2>, write <file2>, delete <file2>.

When a file is opened in (ANSI-)C [21] it must be specified whether the file is to be read, rewritten (written), updated (read and written), or appended to (which is just a particular form of updating; the file is "read" to the end and written from there).

Possible conflicts

- Read/write conflicts occur when something wrong is written into a file due to the reading of (a replica of) another file that is out-dated (or stale). These conflicts are not easily detected, since the system seldom knows how to relate reads and writes. It is easy to detect the possible use of stale date, but not to relate it to following updates. For this purpose transactions can be of great value; operations within the same transaction are logically bound together, and the system knows it because it has been told. The resolution of these conflicts are easy—redo the reads and thereafter the writes based upon them.

- Write/write conflicts occur when the same (logical) file is being updated concurrently in to different places (i.e., two replicas). These conflicts are (relatively) easily detected; comparing (vector) timestamps or file contents. The resolution of these conflicts are not always easy—which updates are the "correct" ones?

3.1.3 Operations on Directories

Operations on directories:

- Inserting, removing, reading, creating, and deleting: The basic operations on directories. Reading is also called listing (the content of) a directory. The entries in a directory are files (or just filenames). (Note, that insert (into directory) is equal to create or link file, and remove (from directory) is equal to delete or unlink file).

- Updating is used for either inserting and/or removing.

As with files, directories can be copied, renamed, and moved. Furthermore, some operations can be combined to include both files and directories, e.g., when moving files from one directory into another.

Possible conflicts [27]

- Update/update conflicts: When different files have been created and/or deleted from the same (logical) directory. This conflict is easily detected and easily resolved; list the two directories to see if they have the same entries, if they do not then perform the (same) operations on the other directory both ways around.

- Name/name conflicts: When two logically different files have been created using the same pathname. This is easily detected (but can be confused for a write/write conflict); two inconsistent replicas.
- Rename/rename conflicts: When the same logical file has been renamed to two different pathnames. This can only be detected if operations are compared, and that would probably require logging of operations.
- Remove/update conflicts: When a file that has been updated in one place has been removed (elsewhere). This is easily detected and the resolution would probably be to keep the updated file, even if this may come as a surprise to the person who initiated the remove.

I will not spend more energy on directory conflict resolution, and no such thing has been implemented. The interested reader is referred to [27].

3.1.4 File Sharing

Studies of file usage patterns in universities using a UNIX system [52, Ch.5.2.1] have shown that normally there is a low level of file sharing. The same studies have also shown that reads are much more common than writes. Putting these two observations together leads to the conclusion that there is a low level of write sharing! Furthermore, it seems that files often are read or written in their entirety, i.e., reads and writes are sequential rather than random.

At DIKU users have home directories, i.e., parts of the total directory structure are devoted to a single user. Normal use of the system is for the user to work alone on the files located in his or her home directory (and subdirectories); no sharing. If the files from a user's home directory are cached on a mobile computer used (only) by that user, then there will be no conflicts—it is unlikely that the user will be working on the same files both on the mobile computer and on the DIKU system at the same time. In Ficus [13] this behaviour is called a *human write lock*, and I think it is the main reason for the success of mobile computing systems using optimistic replica control strategies (see Section 3.3.3) such as Coda and Ficus.

So far, all studies of file usage patterns (at least to my knowledge) have been on UNIX like environments in settings similar to that at universities. If the same measurements where done in other environments then would they yield the same results? Databases for instance will undoubtedly have higher degrees of write sharing if the whole or big parts of (such as tablespaces) the database are looked upon as one file—file types and granularity of replication

are discussed in Sections 3.1.5 and 3.2. Other environments with computer supported cooperative work (CSCW) may also show higher degrees of write sharing. What about users of mobile computers; do they automatically use their computers in the same way they would use stationary workstations?

Coda's assumptions are based on traces done on stationary workstations—how is one to know that users do not change their behaviour when working with their mobile computer? However, Coda has been successful [42].

In Bayou, on the other hand, it is a necessity that the applications are written with mobility in mind, i.e., that they are adapted to this new environment. Thus, it makes no assumptions with regards to the level of write sharing, but it does make assumptions about the ability of application developers to foresee possible conflicts—which I personally find reasonable!

3.1.5 File Sizes and Types

The UNIX file traces referred to in the previous section have also shown that the average file size is less than 10K, and that there are distinct file types and classes.

I do not know if these traces are out-dated. Nowadays most systems have some sorts of multimedia including high-resolution pictures (e.g., JPEGs & GIFs with sizes from 20K to 300K) and movies (e.g., MPEGs & MOVs with sizes from 500K to many MB), and what about a future with generalized objects in an object oriented (OO) world?

Still, I will not make any efforts to support large files nor different types; at first my system should be used for reading e-mails, newsgroups, working on reports, etc. No multimedia for now!

The Odyssey system is an example of a system that tries to integrate different consistency requirements according to the filetype using a notion of *fidelity*, see [43].

3.1.6 File Sharing Semantics

There are different file sharing semantics to be chosen among [52] when designing and implementing a distributed file system. The preferable one is (always) UNIX semantics.[2] In mobile computing this is simply not possible if the clients are to operate on (i.e., do updates to) shared files during disconnected periods. Session semantics seems a reasonable choice, and then

[2]Or something reasonably close to, e.g., NFS.

maybe later extend it with transactions. I choose session semantics.

Note that it is opens and closes that are referred to in the following sections (especially in Section 3.3).

3.2 Granularity of Replication

I will cache whole files, which makes sense, i.e., how would you feel if you found out during disconnection halfway through a file that the rest was missing? "Chunks" give possibility for better use of bandwidth especially with large files (only the needed parts are transferred), but it is harder to work with, i.e., which parts of a file are needed? Under the assumptions made in 3.1.4 and 3.1.5—i.e., files are read or written in their entirety and files are small (10-22K)—whole file replication is a sensible choice. Even more so, when files in a UNIX environment are nothing more—to the system—than a raw stream of bytes.

The alternative would be structured files (or objects) with the structures known (i.e., semantic knowledge of objects [27], [28]) and/or supported by the file system.

Coda and Ficus use whole file replication. Bayou uses a relational database, thus the granularity can be rows of tables (partial replication by means of (updateable!) views [54]), but for know it replicates the full database!

3.3 Replica Control Strategies

"I am not altogether on anybody's *side*,
because nobody is altogether on my *side*"
– Treebeard (Lord of the Rings)

Here I will discuss replica control strategies in the light of a primary copy scheme with a single server and multiple clients.

3.3.1 Pessimistic

My definition of *pessimism* is:

Definition 3.1 *A* pessimistic *replication strategy employed by a client,* C*, with regards to a file,* f*, ensures that* C*'s reads from or writes to* f *are performed on the newest version seen by the server* ***and*** *the updates are guaranteed to reach the server successfully (under normal circumstances).*

Note, giving such guarantees requires some sort of locking (on the server).

The normal definition of a pessimistic replication strategy is that it allows an update—to any (logical) file at any given time—to be made to at most one of the replicas (of the file) *throughout the system.* This is a more restrictive definition than mine; with my definition two clients cannot be pessimistic about the same file, but it is possible for a client to do optimistic updates (see Section 3.3.3) to a file even if the file is locked due to another client being pessimistic (but the optimistic clients are almost certain to fail). This could be the case if a client could not get in contact with the server and decided to do updating anyway (being optimistic).

3.3.2 Strict

My definition of *strict* is:

Definition 3.2 *A* strict *replica control strategy employed by a client,* C*, with regards to a file,* f*, ensures that* C*'s reads from or writes to* f *are performed on the newest version seen by the server.*

The notion of strictness comes form the world of distributed shared memory (DSM) systems [52, Ch.6.3]; and there a read should return the most recent write *throughout the system* (not just at the server). Such guarantees can only be given (under the assumption that users are not willing to wait "forever") tightly coupled systems (such as DSM systems) or in fast local area networks, in bigger systems and in mobile computing network transfer times are too large to make this feasible.

3.3.3 Optimistic

My definition of *optimism* is:

Definition 3.3 *An* optimistic *replication strategy employed by a client,* C*, with regards to a file,* f*, gives no guarantees.*

Note, the version of, f, read or written may be or become inconsistent with the server version and there are no guarantees regarding updates.

Other systems have the same notion of optimism, i.e., that you can read and write to any replica without guaranteed success.

Coda and Ficus uses optimistic replication. Bayou uses a weak consistency scheme (i.e., the system is optimistic).

3.3.4 Multi-Level Consistency

When it comes to keeping replicas mutually consistent, existing systems use either a pessimistic, a strict, or an optimistic strategy.[3] In a system with mobile computers it might be a good idea to have *multi-level* consistency, due to the variation in quality of communication. Multi-level consistency is means that applications can specify whether they want to be pessimistic, strict, or optimistic when using a particular file, and maybe even have varying degrees of pessimism and optimism.

By introducing multi-level consistency applications can *adapt* their behaviour according to the state of their environment (i.e., the characteristics of communication) and/or user demands. They can relax their consistency requirements as the quality of communication decreases in order to achieve higher availability or reduce cost, and strengthen them again when suited.

In a system with multi-level consistency (using session semantics) pessimism will naturally have higher precedence than optimism. If an application uses a file optimistically (e.g., opens it for writing), and then later another application use the same file pessimistically, then the second application will get a lock on the file, and thereby hindering updates done by the first application. Thus pessimism should be used with caution, as it might decrease availability (for other applications/clients than the one being pessimistic).

In Coda, Ficus, and Bayou conflicts are always possible because they use optimistic replication only. In a system with multi-level consistency conflicts can be avoided if it is in the interest of the application (and thereby user). The application decides when to be optimistic (and to what extent) and when **not** to; optimism is not layed down *a priori* by the system.

3.3.5 Conflict Detection

My definition of (mutual) *consistency* is:

Definition 3.4 *A cached (version of a) file,* f'*, is consistent with the primary copy,* f*, if (and only if)* f' *has the update history as* f*.*

If a file is *cached* then the cached version—which is a secondary copy—may or may not be consistent with the server version; the primary copy. Note that with this definition it is not sufficient to have the same content to be

[3] Albeit, under different definitions of the terms!

consistent. Two versions of a file having the same update history should have the same content, whereas two versions of a file having the same content do not necessarily share the same update history. The decision to use update history in the definition rather than content is based on the fact that it is cheaper to compare timestamps than file contents. Since primary copies are stored in one place (a single server) and since consistency is only measured against the primary copy, timestamp comparisons are sufficient to check for consistency, there are no need for version vectors or the like.

To my knowledge Coda, Ficus, and Bayou use similar definitions of consistency. Using a database, as in Bayou [53], allows for testing of logical conflicts, i.e., based on the data in the tables. Bayou applications must specify their own conflict detection, called a *dependency check*, with each write/update.

3.3.6 Conflict Resolution

Conflict resolution is a complicated matter, and I will consider it to beyond the scope of this report. And since conflict resolution is application-specific I will leave it up to the applications, then they can leave the resolution of conflicts to the ultimate experts; the users.

Coda has application-specific resolvers (ASRs) that can be provided by programmers, Ficus has something of the same sort (though I do believe it is somehow based on filetypes) but they are provided by the system. Bayou applications must specify a conflict resolver, called a *merge procedure*, for each write/update.

In Bayou conflict detection and conflict resolution can be given for each write/update, whereas in Coda and Ficus conflict detection is performed by the system and conflict resolvers are global [39]. I think that Bayou's solution is much more flexible, but it inevitably leads to more programming, and existing applications cannot be used at all!

3.4 Replication transparency

When introducing mobile computers into an otherwise stationary distributed system, it must be decided whether the existing applications should continue to function without any changes to them or whether new applications must be developed.

No existing applications (that I am aware of) use multi-level consistency, varying their consistency demands after the characteristics of current communication media. So existing applications must be rewritten (or taken special care of, see Section 5.8) if they are to use such a scheme. It could be decided to develop a whole new environment for mobile computing and write new applications which could make way for an optimal solution. But since the system I am going to develop is to be used in connection with an existing distributed system, then that is really not feasible. A solution that requires only small changes to existing applications is maybe the thing to decide upon, and that I will do.

Coda, Ficus, and MIo-NFS all have replication transparency, that is, the caching of files on the mobile computers, conflict detection, and conflict resolution is handled by system (as much as possible). Bayou on the other hand has absolutely no replication transparency; applications must be written with awareness of the fact that they are working on replicas and supply conflict detection and resolution.

A funny consequence of replication transparency is that when the system fails to resolve a conflict then the resolution of it is **not** left to the application—because it does not know about the replication—but to the users! So this is a case were transparency on application level leads to non-transparency on user level.

3.5 Synchronization

Clocks must be kept synchronized if a timestamp mechanism is to be used. A simple and easy to implement clock synchronization algorithm is the one that goes by the name *Cristian's algorithm.*

The server must also act as *time server* [52] under the assumption that server time is correct time or for the fact that it is important that the clients are synchronized with the server (even if it is out-of-sync with real time). Of course, the algorithm will only work properly if the client does not change media (causing the messages to and from the server to have a large gap in communication time) during synchronization.

Figure 3.2: Cristian's algorithm (running on the client)

```
procedure SynchronizeTime();
var
 T_start,T_end,T_S,TS: time;
begin
 T_start:=GetClientTime;      { start of communication Time     }
 T_S:=GetServerTime;          { request and receive Server Time }
 T_end:=GetClientTime;        { end of communication Time       }
 T_S:=T_S+(T_start-T_end)/2;  { estimated Server Time           }
 TS:=T_S-T_end;               { calculate Time Skew             }
 if |TS|>MaxTS then
  Warning('Time Skew is beyond Max')
 else
  SetClientTime(T_S)
end; {SynchronizeTime}
```

An explanation of the different "times" used in the above algorithm can be found in Table 3.1. These will also be used in later chapters.

Table 3.1: Times

T	real Time	
T_C	Client Time	
T_S	Server Time	
TS	Time Skew	TS=T_S-T_C

3.6 Caching

This section contains a discussion of which files to be cached and when!

Which files should be cached:

- The ones (recently) referenced,
  i.e., the *current working set* [34] as in LITTLE WORK [15].
- The ones (supposedly) needed,
  i.e., the *critical set* [34] as in Coda [22], [42].

- The ones (most often) used,
  i.e., the *average set* [34].

- Them all,
  i.e., the *full set* as in Bayou [53].[4]

The SEER [26] predictive caching system tries to cache the critical set as well as the average set by observing user behaviour (e.g., logging file references) and computing a *reference distance* between files as a measure of how closely-related files are (pair-wise). It then combines this information with *least-recently-used*, *LRU*, information and user-specified hints.

If the cache size is big enough it might be the full set. If the cache size is limited, a choice between the other three must be made. I choose to cache the current working set, since that solution is the one easiest implemented (in a simple LRU fashion). A choice of the critical set would have resulted in some loss of transparency, because it requires some sort of *user-assisted cache management* [45] such as the *hoarding* facility in Coda [22], [42] or as the user-specified hints in SEER [26]. In order to cache the average set, some sort of mechanism to collect usage statistics would have to be deployed, e.g., a *spying* agent as in D-NFS [7].

The drawback of choosing the simple solution is that the other solutions are very likely to result in fewer cache misses during disconnection—especially Coda has experienced high availability in disconnected mode due to the hoarding mechanism [42]. On the other hand, my choice of a simple LRU solution makes it possible for me, at a later stage, to change my mind. Programs to compute a *priority list* of the files in a critical set (e.g., using an algorithm similar to Coda's *hoard walking*) or an average set could be used. Once a priority list had been computed, the files simply needed to be cached in reversed priority order (lowest priority first), e.g., by opening and closing them for reading one at a time, making the LRU method keep the files with highest priority in the cache. Probably not a very efficient (nor especially transparent) way of *heating up the cache* [16] or doing *demand hoard walking* [42] (there is plenty room for improvement here), but it *is* possible. If the full set is desired then the priority list should simply contain all files (and the cache should be large enough).

[4] The Bayou people are planning to use partial replication [54], in which case—I think—it will be the critical set.

3.7 Summary

This is an overview of the decisions made in the previous sections of this chapter:

1. Only *shared* files and applications are supported.
2. Assumptions made regarding file usage:
   - Low level of write-sharing, especially due to *human write locks*.
   - Small files, average size 10-22K.
   - No special support for different file types.
     - **Note**: Even though I will not make many explicit choices based on these assumptions, the implementation will most certainly function less than optimal if actual system use deviates a lot from these.
3. Session semantics were chosen for file sharing.
4. Granularity of replication is whole files, a sensible choice in light of the above assumptions.
5. A replica control strategy based on a primary copy scheme:
   - Multi-level consistency based on definitions of pessimistic, strict, and optimistic replica control strategies.
   - Conflict detection based on timestamping.
   - Conflict resolution left to the applications.
6. The file system should require minimal changes to existing applications.
7. The clients are assumed to be synchronized with the server, but a simple service to check this, based on Cristian's algorithm, will be provided.
8. Caching is done in a simple least-recently-used (LRU) fashion.

Chapter 4

Transactions

The concept of *transactions* has its origins in database systems (see [5, Ch.16], [47, Ch.6.9]). A transaction is a collection of operations that form a (single) *logical unit of work*. The classic example of a (database) transaction is the transferring of a certain amount of money from one bank account to another, see Appendix B. The essential idea of a transaction is *atomicity*, i.e., either all the operations of the transaction are performed or none of them are performed (there is no middle way).[1]

In the following sections I will look at properties of transactions and their implications (Section 4.1) especially with regards to mobility, and how they can be implemented (Section 4.2).

4.1 Properties of Transactions

In the world of (distributed) operating systems transactions normally have four properties: *atomicity*, *consistency*, *isolation*, and *durability* (*ACID*). As far as I know, the only transactions that have been designed for mobile computing do not uphold all of these properties, in fact they only guarantee isolation (see [30], [31]) which is "funny" since that is the only property of the four properties mentioned that is normally **not** provided with database transactions.[2]

In the following three subsections each of the CID properties will be treated separately, constantly comparing them to properties of database

[1] That a transaction is atomic must not be understood as if it in some sense is the smallest unit, i.e., that it does not consist of distinguishable parts (operations).

[2] In Oracle, for example, you must use explicit locking in order to avoid conflicts, see Section 4.1.1.

transactions. The subsection on isolation will discuss the *isolation-only transactions* (*IOT*s) in more depth. The A for atomicity has already been treated, its the "all-or-nothing" property:

> *In my point of view it makes no sense to discuss transactions that are not atomic—why are they called transactions?*

One can either agree with me on this matter or not. If one does not, then the rest of this chapter will be of no use.

Some applications may find it useful to *nest* transactions, even though by allowing this it is impossible to uphold all of the four ACID properties for both *inner* and *outer* transactions, see Section 4.1.4.

Warning: When the term *transactions* is used in connection with databases it is understood, that they are atomic, consistent, and durable (but **not** necessarily isolated), see, e.g., [5]. In the operating system world the terms *transactions* [4], [34] and *atomic transactions* [46], [47], [52] (implying that there can exist non-atomic transactions) are used interchangeably and it is normally understood (in both cases) that they have the ACID properties. Furthermore, I am not convinced that there is agreement on the use of the term *consistency*, which I will return to in Section 4.1.1.

Unless otherwise stated I will assume that *transactions* have the ACID properties (as these are defined below), and that *database transactions* are atomic, durable, and consistent.

4.1.1 Consistency

As I see it, there are two significantly different levels of consistency to be discussed: *operation level consistency* and *system level consistency*.

Operation Level Consistency

Operation level consistency means that the operations of the transaction do not leave data (or whatever type of object that is operated upon) in an inconsistent state. More precisely, *a transaction, when executed alone, transforms an initially correct system state into another correct state* [6].

For instance, the bank account transaction to transfer \$100 from one bank account to another, may not withdraw \$100 from the first account and then incidentally deposit \$101 on the second account, although this could easily be the case, e.g., due to a programming/typing error. I agree with [4, p.360] that (this type of) consistency generally is "the responsibility of the programmers", thus transactions are assumed to be correct. You could have

consistency constraints [34] (or *integrity constraints* [6]), but I find it unlikely that it is possible to guard against *any* programming error; just think of one or more consistency constraints to avoid the extra $1 deposited in the scenario just mentioned. This view is supported by [6] (in a database context):

> "In practice, a complete specification of the constraints governing a small database is impractical (besides, even if it were practical, enforcing the constraints would not be)."

It is also the way that the ORACLE database works [37]:

> "A transaction should consist of all of the necessary parts for one logical unit of work — no more and no less. Data in all referenced tables should be in a consistent state before the transaction begins and after it ends."

Making operation level consistency the programmers responsibility is not going to make me feel the least bit guilty (just handing the problem over to somebody else), because it is not hard to program in a operation level consistent manner, due to the atomic property of transactions.

Furthermore, it is my belief that consistency constraints are "easier" to specify and check in databases where data is typed, than it must be with shared files that to the operating system normally are nothing but uninterpreted sequence of bytes [52, p.246]. Even though this is the case, the operation level consistency is normally the C in ACID, i.e., it is a property of transactions in (distributed) operating systems. To my knowledge the consistency for database transactions have never been anything but system level consistency.

System Level Consistency

System level consistency is a promise given by the transactional system that the execution of transactions that uphold certain requirements (as a minimum they should be correct, i.e., operation level consistent) leave an initially correct system state into another correct system state.

If the system guarantees that transactions are atomic, isolated, and durable then system level consistency is no more than operation level consistency—if transactions that produce correct result when executed alone, are executed as if they were alone, then they **will** produce correct results.

If, for instance, the system does not guarantee isolation, additional requirements[3] may be put on the transactions for the system to make any promises. In ORACLE transactions are not isolated and additional locking is required [37]:

> "SELECT FOR UPDATE [that acquires explicit row locks] is recommended when you need to lock a row without actually changing it. For example, if you intend to base an update on the existing values in a row, you need to make sure the row is not changed by someone else before your update."

Conclusion

I will assume that transactions are operation level consistent—it is the responsibility of the programmers. System level consistency is a must, but it is necessary to decide which additional requirements (if any) the transactions must fulfill in order to guarantee that. I will return to this matter.

4.1.2 Isolation

Running transactions isolated from each other is to say that the execution of several transactions concurrently produces the same database state as some serial execution of the same transactions; the execution is *serializable* [6].

If transactions were not isolated then transactions that would maintain operation level consistency in a single-user, single-process system (i.e., in a system where transactions would be forced to run one after the other, serialized) could not be guaranteed to maintain operation level consistency in a multi-user, multi-process system. For instance, the two transactions in Figure 4.1 (p. 43) would maintain operation level consistency, if they were isolated, but in the figure they are not, so they do not. I personally think that this is the reason why isolation is considered to be of great importance for transactions in (distributed) operating systems.

[3] In addition to being operation level consistent.

Figure 4.1: Non-isolated database transactions

User 1	User 2

`DECLARE`
`  amount MONEY;`
`BEGIN`
`--Transfer $100 from <A> to <B>`
`  SELECT Balance`
`   INTO  amount`
`   FROM  BankAccounts`
`   WHERE Id=<A>;`	

	`/* Withdraw $100 from <A> */`
	`UPDATE BankAccounts`
	` SET   Balance=Balance-100`
	` WHERE Id=<A>`
	` AND   Balance>=100;`
	`COMMIT;`

`  IF amount>=100 THEN`
`    UPDATE BankAccounts`
`     SET   Balance=amount-100`
`     WHERE Id=<A>;`
`    UPDATE BankAccounts`
`     SET   Balance=Balance+100`
`     WHERE Id=<B>;`
`  END IF;`
`  COMMIT;`
`END;`
`/`	

Note: Assume that user 1 sees a Balance of $100 (i.e., amount=$100) then these two interleaving transactions would result in a Balance of -$100, which were not the intention of neither of the users.

User 1 needs to put a *lock* on the balance of bank account <A>, so that user 2 cannot alter it. In ORACLE you acquire such a lock by selecting for update, e.g., by adding the line `FOR UPDATE OF Balance` to the SELECT statement. Doing so would prevent user 2 from doing an update of the selected data, i.e., the update would simply be forced to wait until user 1 released the lock (by the COMMIT), and then it would fail (i.e., Balance=$0).

The problem could be solved (or more precisely, laid in the hands of the programmers) with the use of locks (see the note below Figure 4.1, p. 43). Few distributed operating systems have explicit locking, so they *must* serialize their transactions. Fortunately, I plan to have locking in my file system (because pessimism requires locking, see Section 3.3.1). I will put the same additional requirement on transactions as ORACLE does, i.e., that additional locking is required in order to guarantee system level consistency. This decision is based on the fact that it is (must be) hard to serialize transactions in a system with mobile computers when these might execute their transactions locally, long before they get in contact with any servers (while disconnected, for example).

Isolation-Only Transactions

In order to cater for transactions in Coda, a special type of transactions has been suggested, namely *isolation-only transactions* (*IOT*s), see [30], [31]. This facility has been thoroughly treated in [34] where it is concluded that IOTs are actually *COT*s (*consistency-only transactions*), because the transactions are not isolated from one another, instead the operations of the transactions are checked for consistency (at server commit time, see Section 4.1.3). As mentioned before, I would hesitate to even call IOTs (or COTs) for transactions, because they are not atomic; in my view it would be appropriate to call them a "consistency check mechanism" or similar.

Serial Transactions

In the distributed file system Deceit (that has no support for mobile computers) they use *serial transactions* (*ST*s) which they define as [46, p.22]:

> "An *atomic* transaction [has] two properties: *recoverability* and *serializability*. Recoverability means that the transaction completes or fails entirely. Serializability means that the transactions exhibit behaviour consistent with some total ordering. A serial transaction is a transaction that only provides serializability."

What they call recoverability[4] is what I call atomicity, and serializability is a way to provide isolation in the sense that transactions do not intervene with other transactions performing conflicting file operations (which is the total ordering they mention). So in fact STs are (to some extent at least) IOTs!

[4]Even though a (serial) transaction partly completes, it might still be possible to recover, as in undo, from it's actions (manually).

4.1.3 Durability

When a transaction is finally committed (successfully), then the changes brought about it should be persistent. In other words the doings of the transaction can only be changed (or undone) by executing new commands or transactions that operate on the "results" of the committed transaction.

In mobile computing it may take a while before the changes made by a transaction reach the server and are (finally) committed there. In the mean time, it makes sense to let new transactions continue working on the in some sense un-committed data. If the first one at a later time is found out to fail then the "following" transactions must fail to, of course. However until that time the doings of a transaction may seem committed to new transactions. This leads to the following definition of *pending* transactions: A transaction is pending or *tentative* until it has been committed *on the server*, though it may appear to be committed from a clients, e.g., a disconnected ones, point of view.

In mobile computing we then have transactions executing and committing (or rolling back) in two stages; on a client and on the server. So an implementation would have to consider the stages of executing, committing, and rolling that are listed in Tables 4.1 and 4.2.

Table 4.1: Different stages for execution and commit of a transaction

LET	Local Execution-Time	*execution-time* in [34]
LCT	Local Commit-Time	*tentative* commit on client
SET	Server Execution-Time	*re-execution* on server
SCT	Server Commit-Time	*commit-time* in [34]

Table 4.2: Different stages for rollback of a transaction

LER	Local Execution Rollback	during local execution (LET)
LCR	Local Commit Rollback	at local commit time (LCT)
SER	Server Execution Rollback	during server (re-)execution (SET)
SCR	Server Commit Rollback	at server commit time (SCT)

4.1.4 Nesting

The idea of nesting transactions, i.e., having transactions inside transactions, is not from the database world. It gives rise to the following definitions:

Definition 4.1 *An* inner *transaction begins and ends within another transaction. An* outer *transaction has an inner transaction. An* innermost *or* bottom-level *transaction is an inner transaction that is not an outer transaction. An (or the)* outermost *or* top-level *transaction is an outer transaction that is not an inner transaction.*

Note: A transaction that is neither an inner nor an outer transaction is the same as a transaction without nesting.

I do not think nesting is such a great idea, because it necessarily violates two of the fundamental properties of transactions, namely atomicity and durability. If nesting of transactions is used then atomicity and durability cannot be hold for both outer and inner transactions:

- If inner transactions are durable, then outer transactions cannot be atomic, i.e., they cannot rollback the changes of the inner transactions, because they are permanent (durable).
- If outer transactions are atomic, then inner transactions cannot be durable, i.e., they cannot keep their changes at a rollback of the outer transactions, because they are atomic.

Conclusion

No nesting, it is just a nice feature that you can easily do without, and it blurs the definitions of atomicity and durability.

4.2 Concurrency Control

"Success generally depends upon knowing
how long it takes to succeed."
– Charles Louis de Secondat Montesquieu

After looking at the properties of transactions, we now turn to the matter of how to implement the transactions, the name of the game is *concurrency control*.

A *strict transaction* is a transaction that consists entirely of strict operations (see Section 3.3.2). I will leave the definition of *pessimistic transactions* and *optimistic transactions* to the reader.

Different techniques, such as Two-Phase Commit, Logging, and Optimistic Concurrency Control, for concurrency control has been thoroughly discussed in [34]. Instead of repeating it here I will present my solution that is based on Optimistic Concurrency Control.

4.2.1 Optimistic Concurrency Control

I suggest a new optimistic concurrency control mechanism based on private workspace and modification times, i.e., instead of checking what the other transactions are doing at commit-time (as in Kung and Robinson's Optimistic Concurrency Control algorithm [29]). I wish to check only the state of the files, see Figure 4.2 (p. 48).

The algorithm does not *serialize* the transactions on the server, i.e., at SCT, but since I have decided not to provide isolation that is not a problem. Neither step 1 nor step 2 in the algorithm could lead to ROLLBACK, if the transaction was pessimistic—the files would have been locked by that transaction, and thus could not have been changed not locked by other operations or transactions!

4.2.2 Boundaries of Transactions

In the database world a (new) transaction is begun (implicitly) after a commit or after a rollback. The commit or the rollback may be performed explicitly (using a COMMIT or a ROLLBACK statement) or implicitly by the system when a program terminates,[5] where a commit is performed on successful termination and a rollback otherwise. There are no explicit BEGINs or ENDs to outline database transactions, and database transactions cannot be nested.

In connection with a file system transactions have to be outlined explicitly, because transactions are not a natural part of the system. Begins, ends, and probably also new transactional operations need to be introduced, so that the system can be told which operations belong to which transactions, and when the transactions are begun and ended.

[5]The "program terminates" should not be taken literally, it could be at termination of any sequence of operations.

Figure 4.2: A new algorithm for optimistic concurrency control

`begin`

<For each file read or written, or until ROLLBACK, do the following:

1. has the file been changed (on the server) so that it is inconsistent with the version the transaction operated on, then ROLLBACK
2. has the file been locked (explicit by pessimistic read or write operations, or implicit by another transaction), then ROLLBACK
3. if neither 1. nor 2. is the case then put a lock on the file

>

<If all files operated on have been successfully locked through the steps taken above, then:

- make a copy of all the files (in case something goes wrong)
- alter all the updated files
- see to that all changes are written to stable storage
- release all the locks, thereby completing the COMMIT

>

`end`

4.3 Summary

This is a summary of the decisions taken and conclusions made in the previous sections of this chapter:

- Transactions are *atomic*.
- Transactions are assumed to be *operation level consistent*. It is the programmers' responsibility to write them that way.
- Transactions are **not** guaranteed to be isolated, so additional locking may be required to guarantee *system level consistency*.
- Transactions are *durable* after having been committed on the server.

- Transactions that are *pending* seem committed to a client.
- Transactions cannot be nested.
- A new algorithm for optimistic concurrency control was sketched.
- Explicit begins and ends, and new transaction operations are needed.

Chapter 5

The Model

I have decided that the implementation takes place at the C [21] interface level (e.g, replacing `fopen` and `fclose`), not at the UNIX system level (e.g., neither `open` nor `close`). This will make it possible to use the system on other platforms (assuming that the programmers use the C interface there). However it might result in even more overhead than has probably already been introduced by implementing the C interface on top of the UNIX primitives—you win some, you loose some. I have decided to do so anyway, simply because it eases the implementation.

I have already decided to do LRU-caching, see Section 3.6, but I have not yet decided whether the system should cache multiple copies of a file—i.e., if multiple processes on the client are using the file—or just one copy.

I have decided to cache just one copy.[1] If the user of the mobile client is using programs that access files concurrently (e.g., multiple instances of the same program), then he or she might end up with local conflicts—but *it is* probably the users own fault, since *mobile computers are normally in use by a single person at a time*, see Sections 1.2.1 and 2.3. Thus it is really no step back compared to what the users in a multi-user environment are faced with, because there the fault might as well be some other users.

This decision does, however, conflict with the traditional definition of *session semantics* [3]. My system guarantees session semantics pr. client and **not** pr. application.

Much of the presentation that follows is based on an article [51] presented at the OOPSLA'96 Workshop on Object Replication and Mobile Computing (ORMC'96).

[1] This decision is new compared to the contents of [51].

Recall, that I am using a primary copy scheme with a single server and multiple clients using session semantics. The model is based on the use of time as a consistency measure. With each cached file is associated:

- A Modification Time (MT_C); the time of the last update to the file (on the server),
  - **Note**: From the subscript it is made clear that the replica resides on the client (as opposed to on the server).
- a Consistency Time (CT); the time at which the cached file was known to be consistent with the primary copy on the server, and
- a Consistency Check Time (CCT); the time of the last check for consistency between the cached file and the primary copy on the server.

If $CT = CCT$, then the last check for consistency was positive, otherwise negative. According to the definition of consistency, see Definition 3.4 in Section 3.3.5, a consistency check can be performed simply by comparing the modification time (MT_C) of the cached file with the modification time (MT_S) of the primary copy on the server.

With every open for reading must be associated a Consistency Time Bound (CTB), and with every open for writing must be associated a Modification Time Bound (MTB). Furthermore, an Expiration Time Bound (ETB) must be associated with each close. These time bounds must be given explicitly by the application. Their exact meanings are explained in Sections 5.1, 5.2 and 5.5, respectively.

Throughout the rest of this chapter the notations listed in Tables 5.1, 5.2, 5.3 will be used extensively; they will be explained later.

Table 5.1: Consistency times

CT	Consistency Time	
CCT	Consistency Check Time	
CF	Consistency Flag	$CT = CCT \Leftrightarrow CF$
MT	(Server) Modification Time	

Table 5.2: Expiration times

ET	Expiration Time
RET	Readlock Expiration Time
WET	Writelock Expiration Time

Table 5.3: Time bounds

CTB	Consistency Time Bound
MTB	Modification Time Bound
ETB	Expiration Time Bound (*)
RETB	Readlock Expiration Time Bound
WETB	Writelock Expiration Time Bound

(*): Referred to as Close Expiration Time (*CET*) in [51].

5.1 Reading

I feel that the model is best presented by means of examples, so here goes! This is the example used in [51].

Imagine a client holding a replica of a file, `f`, in the cache with the associated values $<MT_C$=8.00,CT=9.00,CCT=9.00>. This would be the result if the file was last updated (before 9.00) on the server at 8.00, and cached on the client in question at 9.00. At 10.00 (being now) the client opens the file for reading with a command similar to:

`open(f,READING,`*CTB*`)`

5.1.1 Pessimistic Reading

If *CTB*>0 then `f` must be and remain consistent (with the server version) within the specified time bound. For example, with *CTB*=2 hours (from now), then `f` must be guaranteed to be and remain consistent until 12.00. In other words, `f` must not have been updated on the server between 9.00 (where it was cached) and 10.00 (which is now), and furthermore, it must not be updated on the server (by other clients) for the next two hours.

If `f` has been updated between 9.00 and 10.00 then a new copy of it is required, for it to be consistent now. Under any circumstance a readlock

must be obtained, for it to remain consistent within the next two hours. Locking of files are dealt with in Section 5.4.

If obtaining a new copy of `f` or a readlock on it (or both) succeeds then the `open` succeeds, otherwise it fails.

Using $CTB>0$ the client is *pessimistic*. The greater the CTB the higher the degree of pessimism.

5.1.2 Optimistic Reading

If, on the other hand, $CTB<0$ then the client is satisfied with `f` if it was consistent at least no longer ago than specified by the time bound. In the example with CTB=-2 hours (back), the read succeeds because `f` was consistent sometime within the last two hours, namely one hour ago at 9.00.

With CTB=-$\frac{1}{2}$ hour, a new consistency check is required because `f` cannot be guaranteed to have been consistent half an hour ago at 9.30. If it has not been updated on the server (by another client) since 9.00 then it (the cached file) can be used, otherwise a new copy is required (either way CT and CCT can be set to 10.00). If obtaining a new copy is necessary and that fails then the `open` fails.

Using $CTB<0$ the client is *optimistic*. The more negative the CTB the higher the level of optimism.

5.1.3 Strict Reading

If, finally, $CTB=0$ then `f` must be consistent now, but is not guaranteed to remain consistent. As in the previous cases a new version of `f` may be required.

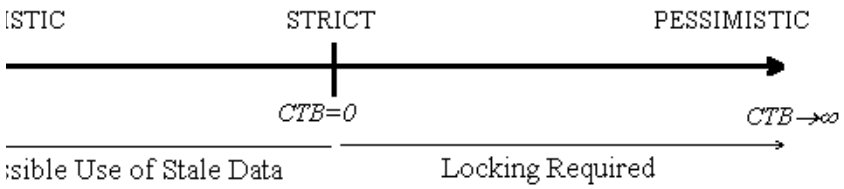
Using $CTB=0$ the client is being *strict*.

5.1.4 The Consistency Time Bound

The correspondence between the consistency time bound and the level of optimism or pessimism when opening a file for reading is depicted in Figure 5.1.

It should be noted that the higher the level of optimism by one client the lower the level of availability for other clients!

Figure 5.1: Consistency Time Bound (CTB)



To sum up the previous sections:

When a file, `f`, is opened for reading and *CTB* is the specified consistency time bound, then the following "rules" apply:

- If *CTB*>0 then `f` must **be and remain consistent** within the specified time bound. This is *pessimistic reading*. With an increasing *CTB* the level of pessimism increases.
- If *CTB*=0 then `f` must **be consistent** (now). This is *strict reading*.
- If *CTB*<0 then `f` is guaranteed to **have been consistent** no longer ago than specified by the time bound. This is *optimistic reading*. With a decreasing *CTB* there is an increasing chance of reading stale data (inconsistent files).

CTB should be thought of as a system guarantee, and the user or the application may end up doing better—e.g., if available bandwidth is used intelligently!

Even though we previously assumed that the mobile clients did not stay disconnected forever (see Section 1.3.1), this is in fact possible. All cached files can be read at any time; using an infinitely negative *CTB*!

The primitive for open for reading is the ANSI-C [21] `fopen` with an added parameter for specifying the consistency time bound:

- ANSI-C: **FILE *fopen(char *pathname,char *mode);**
  - **mode** $\in$ {**"r","rb"**}
- $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$: **FILE *p_open(char *pathname,char *mode,int ctb);**

5.2 Writing

The *MTB* associated with an open for writing is the "mutating" counterpart of the *CTB* associated with an open for reading. The command is similar:

`open(f,WRITING,`*MTB*`)`

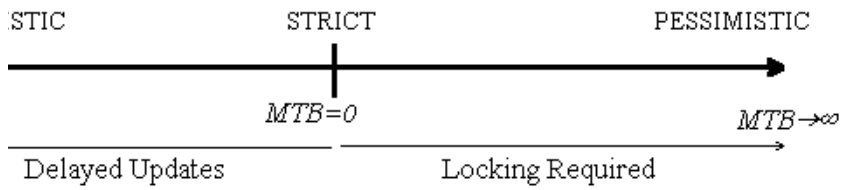
When a file, `f` (as in the above command), is opened for writing and *MTB* is the specified time bound, then the following "rules" apply:

- If *MTB*>0 then `f` must **be and remain consistent** within the specified time bound. This is *pessimistic writing*.
  - **Note**: In this case *consistency* should be thought of as: The file is not updated on the server by other clients. The replica, on the client that opens the file for writing, of course do not stay consistent (because it is written to), but it will "merge" successfully with the primary copy if it reaches the server before the lock expires—so in this sense it *is* consistent.
- If *MTB*=0 then `f` must **be consistent** (now). This is *strict writing*.
- If *MTB*<0 then `f` is guaranteed to **have been consistent** no longer ago then specified by the time bound **and** on last check. This is *optimistic writing*.
  - **Note**: Compared to reads the file is additionally required to have been consistent on the last check (i.e., *CT*=*CCT*), for else the writing is bound to fail (it is inconsistent even before it is begone).

As with the *CTB* the *MTB* should be thought of as a system guarantee and the user or the application may end up doing better—e.g., if available bandwidth is used intelligently!

The correspondence between the modification time bound and the level of optimism or pessimism when opening a file for writing is depicted in Figure 5.2.

Figure 5.2: Modification Time Bound (MTB)



The primitive for open for writing is the ANSI-C [21] `fopen` with an added parameter for specifying the modification time bound:

- ANSI-C: **FILE *fopen(char *pathname,char *mode);**
  - **mode** $\in$ {**"r+","r+b","w","wb","w+","w+b","a","ab","a+","a+b"**}
- $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$: **FILE *p_open(char *pathname,char *mode,int mtb);**

5.3 Creating & Deleting

Creating files is done by opening non-existent files for writing. I have decided to provide a primitive for removal of a file that checks for any locks and also (if possible) deletes the primary copy on the server. The implementation is done in the simplest way; the removal fails if the server cannot be reached.[2]

The primitive for removing is the same as the ANSI-C [21] `remove`:

- ANSI-C: **int remove(char *pathname);**
- $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$: **int p_remove(char *pathname);**

5.4 Locking

Introducing locks into a system inevitably leads to a lot of decisions that must be made. To cut things short, I have decided to go with a very simple solution:

1. Locks are non-blocking.
2. All locks must be accepted by the server.
3. A file can have multiple readlocks (by the same number of clients) or a single writelock (by a single client).
4. A client can obtain at most one lock pr. file.
5. A client cannot renew a previously obtained lock for additional time.

[2] A harder way of doing it (but more in line with the rest of the model), would be to let the application specify a modification time bound; removal is a mutating operation.

6. A client cannot convert a readlock to a writelock or vice versa.

7. Locks that are obtained in order to read **and** write are treated as writelocks.

Decision 1 renders it unnecessary to implement waiting queues or the like, but it also lessens the possibility of deadlocks (now programming deadlocks are left entirely to the programmers). Decisions 2 and 3 are obvious. Decisions 4 to 7 reduce the complexity of the implementation, but they also reduce server traffic—a client knows if it already has a lock on a file, and thus do not need to contact the server to learn that a new lock request must fail. Furthermore, decisions 5 and 6 give fairness, and thus no starvation. Schematic overviews of the outcome of performing a readlock, a writelock, a readunlock, and a writeunlock are given in Figures 5.4, 5.5, 5.6, and 5.7. The decisions admittedly give little flexibility, but on the other hand the locking mechanism is easily understood.

Note, that the decisions naturally reflect back on pessimistic read and write opens, since they require locking.

I have decided to provide primitives for locking and unlocking files. These may come in handy when a user does a voluntary disconnection, knowing that he or she is going to update the file later.

If a client locks a file that is not cached on the mobile client, then the file gets cached. This is done under the assumption that if the client locks the file (e.g., in a situation like the one just mentioned), then it **is** going to be used.

In order to provide an interface that is similar to that of the other file operation primitives I have decided upon these two file locking primitives:

- **int p_lock(char *name,char *mode,int etb);**
- **int p_unlock(char *name,char *mode);**[3]

[3]Since it has been decided that a client can obtain at most one lock pr. file, the `mode` parameter is actually not needed—the system knows which kind of lock that the client has on the file. However, at the time this was pointed out to me [3], it had already been implemented (and partly tested). It is not really that bad, because the client should **also** know which kind of lock that has been put on the file!

Table 5.4: Performing a read lock

REQUEST: `p_lock(name,"r",`*RETB*`)`			
STATUS BEFORE	OUTCOME	STATUS AFTER	S
`NOT_LOCKED`	success	`READ_LOCKED` by yourself, *RET*=T_C+*RETB*	*
`READ_LOCKED` by yourself	failure	`READ_LOCKED` by yourself, *RET* unchanged	
`READ_LOCKED` by others	success	`READ_LOCKED` along with others, *RET*=T_C+*RETB*	*
`READ_LOCKED` along with others	failure	`READ_LOCKED` along with others, *RET* unchanged	
`WRITE_LOCKED` by yourself	failure	`WRITE_LOCKED` by yourself, *WET* unchanged	
`WRITE_LOCKED` by another	failure	`WRITE_LOCKED` by another	*

Note: The S-columns indicates with an asterix when it is necessary to contact the server. If the server is unreachable in these cases then the outcome is failure.

Table 5.5: Performing a write lock

REQUEST: `p_lock(name,"w",`*WETB*`)`			
STATUS BEFORE	OUTCOME	STATUS AFTER	S
`NOT_LOCKED`	success	`WRITE_LOCKED` by yourself, *WET*=T_C+*WETB*	*
`READ_LOCKED` by yourself	failure	`READ_LOCKED` by yourself, *RET* unchanged	
`READ_LOCKED` by others	failure	`READ_LOCKED` by others	*
`READ_LOCKED` along with others	failure	`READ_LOCKED` along with others, *RET* unchanged	
`WRITE_LOCKED` by yourself	failure	`WRITE_LOCKED` by yourself, *WET* unchanged	
`WRITE_LOCKED` by another	failure	`WRITE_LOCKED` by another	*

Table 5.6: Performing a read unlock

REQUEST: `p_unlock(name,"r")`			
STATUS BEFORE	OUTCOME	STATUS AFTER	S
`NOT_LOCKED`	failure (!)	`NOT_LOCKED`	
`READ_LOCKED` by yourself	success	`NOT_LOCKED`	*
`READ_LOCKED` by others	failure	`READ_LOCKED` by others	
`READ_LOCKED` along with others	success	`READ_LOCKED` by others	*
`WRITE_LOCKED` by yourself	failure	`WRITE_LOCKED` by yourself, *WET* unchanged	
`WRITE_LOCKED` by another	failure	`WRITE_LOCKED` by another	*

(!): Even though this is what is wanted it must be considered a fault.

Note: The S-column indicates with an asterix when it is necessary to contact the server. If the server is unreachable in these cases then the outcome is failure.

Table 5.7: Performing a write unlock

REQUEST: `p_unlock(name,"w")`			
STATUS BEFORE	OUTCOME	STATUS AFTER	S
`NOT_LOCKED`	failure (!)	`NOT_LOCKED`	
`READ_LOCKED` by yourself	failure	`READ_LOCKED` by yourself, *RET* unchanged	
`READ_LOCKED` by others	failure	`READ_LOCKED` by others	*
`READ_LOCKED` along with others	failure	`READ_LOCKED` along with others, *RET* unchanged	
`WRITE_LOCKED` by yourself	success	`NOT_LOCKED`	*
`WRITE_LOCKED` by another	failure	`WRITE_LOCKED` by another	*

(!): Even though this is what is wanted it must be considered a fault.

5.5 Conflicts

Since optimistic write operations are allowed, conflict situations are unavoidable. Since I chose session semantics for file operations then conflicts for a given file can only be detected after that file has been closed.

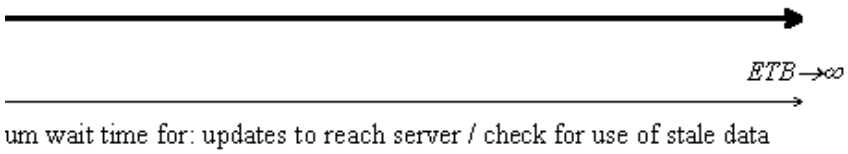
If it takes some amount of time before the updates are propagated to the server, then it is impossible to report failure or success immediately, that is, whether there is a conflict or not. We could let the close operation block until "the matter has been sorted out", but not all applications are geared to do so—they probably expect the close to return immediately. How long an application can wait is specific to that application, so a way of letting the application tell the system, how long it is prepared to wait, is needed. I chose to associate yet another time bound with the close operation (well along the line of the other operations)—the Expiration Time Bound (*ETB*).

Imagine a client closing a file, `f`, by issuing a command similar to:

`close(f,`*ETB*`)`

A `close` command, like the one above, will not return until either all updates have been propagated successfully back to the server, or a conflict has been detected, or it times out (expires) according to the specified time bound (see Figure 5.3). The command can return `SUCCESS`, `FAILURE`, **or** `TIMEOUT`.

Figure 5.3: Expiration Time Bound (ETB)



If the `close` times out then the application can take appropriate actions, e.g., inform the users that they must check `f` later (themselves) because it cannot say whether the `close` eventually succeeds or not![4]

If the `close` results in failure and the file was opened for writing, then the application can inform the user that his or her updates failed, and copy the contents of the cached file to another (new) file (or do whatever actions

[4]It may be a good idea to provide some way for the application to check at a later stage, but the current implementation does not.

it deems necessary). If the file was opened for reading and the `close` fails then it means that the user have been reading stale data (i.e., from a file that has been updated on the server in the meantime); the application can ignore this or, if it is "polite", inform the user.

The primitive for closing a file is the ANSI-C [21] `fclose` with an added parameter for specifying the expiration time bound:

- ANSI-C: **int fclose(FILE *fp);**
- $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$: **int p_close(FILE *fp,int etb);**

The two following subsections are dedicated to the matters of how conflicts are best avoided, when they are detected, and how they can be resolved! First subsection is about *read/write conflicts* and the other about *write/write conflicts*.

5.5.1 Read/Write Conflicts

In this section I will look at read/write conflicts, i.e., writes based on or using out-dated data.

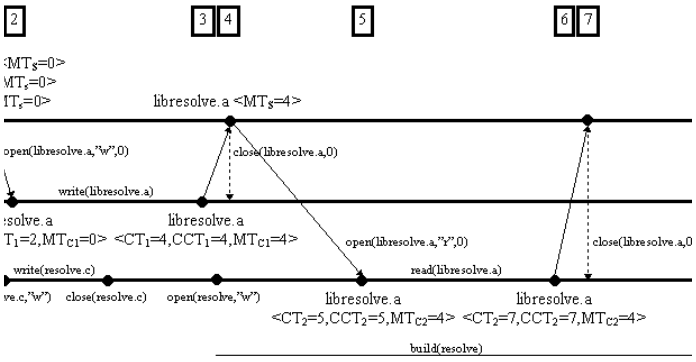
Example Scenario

- Bill using client machine no. 1 opens `libresolve.a` (strict, i.e., MTB=0) for writing, does his updating (locally), and closes `libresolve.a`.
- Joe using client machine no. 2 opens `resolve.c` for writing, does his updating (locally) and closes `resolve.c`. Then he (re-)builds the program `resolve` with `libresolve.a` linked in, thus `libresolve.a` is opened (strict, i.e., CTB=0) for reading, read (locally), and closed during the build.

Possibility of conflict:

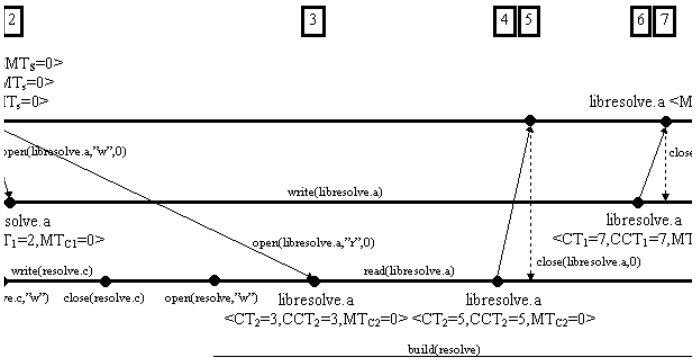
- No read/write conflict:
  If Bill gets all his work done, before Joe starts reading `libresolve.a` then everything is fine, i.e., no read/write conflict occurs. This is depicted in Figure 5.4 (If, contraversely, Joe gets all his work done, before Bill starts rewriting `libresolve.a` then there is no read/write conflict either—Joe used the latest possible update of the library file).

Figure 5.4: No read/write conflict



- Read/write conflict:
  A read/write conflict occur when Joe starts reading (as a side effect of his rebuilding of `resolve`) `libresolve.a` after Bill has started—but before he has finished (or before the updated version has reached the server)—rewriting it, or when Bill starts rewriting while Joe is reading!

Figure 5.5: Undetectable read/write conflict



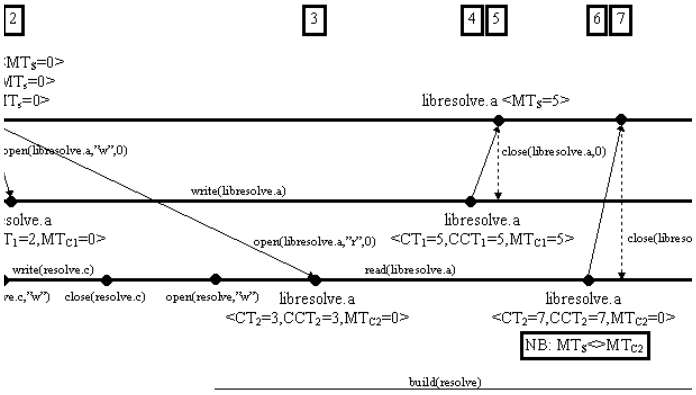
Detection of read/write conflict:

- The read/write conflict depicted in Figure 5.5 could be detected by use of transactions, that is, if Joe had grouped his work into one transaction

that ended after the close of `resolve`. This is hinted by the stippled arrow from client 2 to the server at the `close(resolve)` command. For now, the system cannot detect conflicts of this type.

- One could argue that to contact the server after closing a file opened for reading only is not needed, since it does not affect the server, but it opens up the possibility of detecting some read/write conflicts, i.e., such as the one depicted in Figure 5.6. Therefore I chose to contact the server after a "read" close, to report such a situation.

Figure 5.6: Detectable read/write conflict



Using locks to avoid read/write conflicts:

- Now, if Bill had locked the file for writing (i.e., being pessimistic using $MTB>0$) , the open for reading in Figure 5.5 and the open for reading in Figure 5.6 would have failed, and thus the read/write conflicts avoided—at the expense of letting Joe wait until Bill had finished!

- If Joe had started reading before Bill started rewriting, then read/write conflicts could have been avoided if Joe had been pessimistic (i.e., using $CTB>0$, and thus readlocking `libresolve.a`).

In other words, the introduction of locks into the system, to some extent can make up for the lack of such a thing as IOTs—at the expense of some availability.

5.5.2 Write/Write Conflicts

Write/write conflicts are easily detected. When a client wishes to send an updated file to the server, then the server simply checks the associated modification time (MT_C) with that of the primary copy (MT_S). If $MT_C <> MT_S$ then there is a conflict. Conflicting files are never send to the server. Of course, the server also has to check if the file is locked (by other clients).

The only way for a client to make absolutely sure that updating to a file will succeed is to obtain a writelock on the file, close the file and get in contact with the server before the writelock expires!

5.6 Other Features

5.6.1 Temporary Files

Since temporary files should be handled differently than persistent files (see Section 3.1.1)—because there is no need to "bother" the server with temporary files—it would be a good idea to provide special primitives for these. This way the applications can tell the system which files are temporary (and that is the only way for the system to found out). I have not implemented any of these, because they are already there; in ANSI-C [21].

Primitives from ANSI-C for using temporary files:

- **FILE *tmpfile();**
- **char *tmpnam(char s[L_tmpnam]);**

5.6.2 Synchronization

A synchronization primitive based on Cristian's algorithm (see Section 3.5) is provided.

The $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ synchronization primitive is:

- **long p_time(int flag);**

The `flag` should be used to signal whether the client just wants to get the (estimated) server time, or whether it actually wants to synchronize (i.e., set client time to the estimated server time). A very simple synchronization program is shown in Figure 5.7.

5.6.3 Status

To let applications adapt their behaviour with respect to the (caching) status of a file and the current quality of communication, I have decided upon two primitives that supply these informations.

The $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ status primitives are:

- **int p_stat(char *name,long *mt,long *ct,long *cct,long *ret, long *wet);**
- **int p_comm();**

In Figure 5.8 (though a bit out of place) is an example use of these primitives.

5.6.4 More Primitives

As suggested by the lists of operations on files and directories in Sections 3.1.2 and 3.1.3, there are lots of opportunities for providing additional primitives (a `rename`, a `copy`, a ...). I have decided to do none of these, since most of them can (more or less elegantly) be written using the other (file) primitives.

Figure 5.7: Synchronization

```
/* psync.c - PeStO sample application */

#include "pclient.h"

int main()
{
  long t;

  if((t=p_time(SET_TIME))==(long)NOT_OK) {
    p_perror("Synchronization using p_time failed");
    return -1;
  }
  else {
    printf("Client time set to %s",ctime(&t));
    return 0;
  }
}
```

Figure 5.8: Example use of status primitives

```
/* pstatus.c - PeStO sample application */

#include "pclient.h"

int initstat=TRUE;

void filestatus(name)
  char *name;
{
  int cached;
  long t,mt,ct,cct,ret,wet;

  if(initstat) {
    printf("\n%-30s Cached Consistent Readlocked Writelocked\n","Filename");
    initstat=FALSE;
  }

  if((cached=p_stat(name,&mt,&ct,&cct,&ret,&wet))==NOT_OK)
    printf("%-30s %6s %10s %10s %11s\n",name,"?","?","?","?");
  else
    if(!cached)
      printf("%-30s %6s %10s %10s %11s\n",name,"NO","-","NO","NO");
    else {
      time(&t);
      printf("%-30s %6s %10s %10s %11s\n",name,"YES",(ct==cct) ? "YES" : "NO",
     (ret>=t) ? "YES" : "NO",(wet>=t) ? "YES" : "NO");
    }
}

void communicationstatus()
{
  int commstat;

  printf("\nCommunication Status: ");

  commstat=p_comm();

  switch(commstat) {
    case NOT_OK:
      printf("NOT WORKING PROPERLY\n");
      break;
    case CONNECTED:
      printf("FULLY CONNECTED\n");
      break;
    case WEAKLY_CONNECTED:
      printf("WEAKLY CONNECTED\n");
      break;
    case DISCONNECTED:
      printf("DISCONNECTED\n");
      break;
    default:
      printf("UNKNOWN\n");
  }
}

void main(argc,argv)
  int argc;
  char *argv[];
{
  communicationstatus();

  for(; argc>1;)
    filestatus(argv[--argc]);

  printf("\n");
}
```

5.7 Primitives

Client programs wishing to access shared files using the $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ file system should start with a `#include "pclient.h"`. The client interface (available primitives) is described in the following sections.

5.7.1 File Primitives

The following file primitives are provided:

- `FILE *p_open(char *name,char *mode,int tb)`
  returns a valid filepointer (`FILE *`) or `NULL`.
- `int p_close(FILE *fp,int etb)`
  returns `SUCCESS`, `TIMEOUT`, `FAILURE`, or `NOT_OK`.
- `int p_lock(char *name,char *mode,int etb)`
  returns `OK` or `NOT_OK`.
- `int p_unlock(char *name,char *mode)`
  returns `OK` or `NOT_OK`.
- `int p_stat(char *name,long *mt,long *ct,long *cct,long *ret, long *wet)`
  returns `CACHED` with `mt`, `ct`, `cct`, `ret`, and `wet` set, `NOT_CACHED`, or `NOT_OK`.
- `int p_remove(char *name)`
  returns `OK` or `NOT_OK`

Errors are reported by the `pso_errno` variable. If no errors are encountered `pso_errno` is set to `PSO_ENOERROR`. An error message corresponding to the given error number can be written using the `p_perror` procedure, see Section 5.7.2.

Legal values of `mode` are:

- For reading (only): `"r"`, `"rb"`.
- For (re-)writing: `"w"`, `"wb"`, `"w+"`, `"w+b"`.
- For reading and writing (i.e., updating): `"r+"`, `"r+b"`.
- For appending: `"a"`, `"ab"`, `"a+"`, `"a+b"`.

Note: The additional `b` (for binary) makes no difference in a UNIX environment. All strings containing either `r`, `w`, or `a` are initially accepted by $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ but if they are illegal (i.e., not one of those listed above) with respect to the standard ANSI-C mode used in `fopen` they might fail after all.

Time bounds, `tb` or `etb`, are given in whole minutes. Expiration time bounds, `etb`'s, less than zero are treated as zeros.

5.7.2 System Settings & Primitives

This section lists the possible settings for the system and the few primitives provided by $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ that do not fall into the file operation category.

The system settings:

- `TIMESKEW_MAX` in `"pesto.h"`: This is the maximum time skew. If it is not the case that $|TS| \leq$ `TIMESKEW_MAX` (where TS is the estimated time skew) then the client's clock is drifting too much with respect to the server's clock.
- `LOG` & `LOGFILE` in `"ppserver.h"`: Should the server keep a log of the received requests?

Other possibilities are (but these are not implemented): maximum readlock and writelock expiration time bounds.

The system primitives are:

- `long p_time(flag)`
  returns estimated server time or `(long)NOT_OK`.
- `int p_comm()`
  returns `CONNECTED`, `WEAKLY_CONNECTED`, `DISCONNECTED`, or `NOT_OK`.
- `void p_perror(char *str)`

The `flag` used as parameter to `p_time` should be `GET_TIME` or `SET_TIME`. If `SET_TIME` then the client's time will be set to the estimated server time.

5.7.3 Transaction Primitives

The suggested transaction interface is:

- `transaction_id *begin_transaction(int ctb,int mtb,int etb)`
- `FILE *t_open(transaction_id tid,char *pathname,char *mode)`
- `int t_close(transaction_id tid,FILE *fp)`
- `void abort_transaction(transaction_id tid)`
- `transaction_status end_transaction(transaction_id tid)`
- `transaction_status status_transaction(transaction_id tid)`

Since these have not been implemented I will not use a lot of space on discussing their use. Instead I will give a few notes and give an example of how I imagine them used:

- All reads and writes using the file pointer returned by a transactional open, `t_open(*tid,...)`, are considered part of the transaction identified by `tid`.
- Before an `abort_transaction` all open files that are part of the transaction (i.e., opened using `t_open`) must be closed explicitly.
- The application can test (using `status_transaction`) for the status of a transaction, and re-execution of the transaction is the responsibility of the application—the system does not provide any functionality of such sort. The possible statuses of a transaction is given in Table 5.8.
- Example use of transactions are given in Figures 5.9 and 5.10 pp. 72-73. Please note that these are examples of how I think it could be done—it has not been implemented!
- The optimism, strictness, or pessimism of the transaction is tied to the distributed file service using the same (notions of) consistency, modification, and expiration time bounds.
  - E.g., if the transaction is a pessimistic one (and communication is good enough to get the changes to the server "in time") then `COMMITED_ON_CLIENT` is sufficient; the updates are guaranteed to reach the server due to the obtained write-locks, and the reads are guaranteed to be consistent due to the obtained read-locks. If the locks were not obtained, the transaction would have failed.

- On the server a transaction could be uniquely identified by the transaction id (from the client) and a machine (the client's) id (i.e., IP-number).

Table 5.8: Transaction status values

Status	Meaning
`NOTBEGUN`	Unable to start; `begin_transaction` failed?
`RUNNING_ON_CLIENT`	Transaction is executing on the client
`ABORTED`	Explicitly aborted using `abort_transaction`
`ABORTED_ON_CLIENT`	Attempt to commit on client side failed
`COMMITED_ON_CLIENT`	All transactional statements succeeded on client
`RUNNING_ON_SERVER`	Transaction is attempting commit on the server
`ABORTED_ON_SERVER`	Attempt to commit on server failed
`COMMITED_ON_SERVER`	All transactional statements succeeded

Whether a successful implementation based on these ideas is possible or not, only the future can tell. I rest my case.

5.8 Existing Applications

Existing applications could be ported easily using:

- *pessimistic*: `fopen(name,mode)` → `p_open(name,mode,`∞`)` or using `p_lock(name,mode,`∞`)`
- *strict*: `fopen(name,mode)` → `p_open(name,mode,0)`
- *optimistic*: `fopen(name,mode)` → `p_open(name,mode,-`∞`)`

Note: The ∞ in the above `read`-statements means a sufficiently large number (so big that it will last the whole of the systems lifetime).

Default values for *CTB*, *MTB*, and *ETB* pr. system or application could be used implicitly when running existing applications (unchanged). However this would require new daemons to catch, for instance, access to remote files via NFS, e.g. as it is done in [32].

Figure 5.9: Example use of transactions (1)

```
transaction_status do_transfer(ctb,mtb,etb,tid)
  int ctb,mtb,etb;
  transaction_id *tid;
{
  FILE *fpA,*fpB;
  money amountA,amountB;

  if((*tid=begin_transaction(ctb,mtb,etb))==NOT_OK)
    return NOTBEGUN;

  /* Transactional open -- uses ctb or mtb implicit */
  if((fpA=t_open(*tid,"BankAccA.bal","r+"))==NULL) {
    abort_transaction(*tid);
    return ABORTED;
  }

  /* Read balance on BankAccount A */
  fscanf(fpA,"%10.2f",&amountA);

  if(amountA>=100.00) {
    /* Withdraw $100 from  BankAccount A */
    rewind(fpA);
    fprintf(fp,"%10.2f,"",amountA-100.00);

    if((fpB=t_open(*tid,"BankAccB.bal","r+"))==NULL) {
      t_close(fpA);
      abort_transaction(*tid);
      return ABORTED;
    }

    /* Deposit $100 on BankAccount B */
    fscanf(fpB,"%10.2f",&amountB);
    rewind(fpB);
    fprintf(fpB,"%10.2f",amountB+100.00);

    /* Transactional Close -- uses etb implicit */
    t_close(*tid,fpB);
  }

  t_close(*tid,fpA);
  return end_transaction(*tid);
}
```

Figure 5.10: Example use of transactions (2)

```
void main()
{
  transaction_id tid;
  transaction_status tstat;
  int ctb,mtb,etb;
  long timeout,starttime;
  int retries,stop;

  ctb=mtb=etb=1; /* pessimistic, one minute of locking */

  if((tstat=do_transfer(ctb,mtb,etb,&tid))==NOTBEGUN)
    exit("Unable to initiate transfer");

  retries=5;   /* stop after 5 retries */
  timeout=600; /* timeout after 10 min */
  stop=FALSE;
  starttime=time();
  while(tstat!=COMMITED_ON_SERVER&&retries>0&&!stop) {
    switch(tstat) {
      case ABORTED_ON_SERVER:
        ctb=...; mtb=...; etb=...; /* use new values */
      case ABORTED:
      case ABORTED_ON_CLIENT:
        retries--;
        tstat=do_transfer(ctb,mtb,etb,&tid); /* re-execute */
        break;
      default:
        tstat=status_transaction(tid); /* check status */
    }

    if(tstat==NOTBEGUN||starttime+timeout>time())
      stop=TRUE;
  }

  if(tstat!=COMMITED_ON_SERVER)
    exit("Transfer failed");

  printf("Transfer succeeded\n");
}
```

Chapter 6

The Implementation

This chapter describes the actual implementation in some details. I have used the following sources [48], [12], [23], and [50] for information of how to program client/server applications in a UNIX environment using sockets. Code from all of these can be refound in some form in $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$.

6.1 System Requirements

Without TACO (client and server):

- UNIX System V (`SYSV`) compliant, or
- Linux (`LINUX`) kernel version 1.2.13 (or greater).

With TACO (client and mobility support gateway) [12]:

- Linux kernel version 1.2.13,
- PPP networking support, and
- PC Card services version 2.8.9 by David Hinds (client only).

6.1.1 Test Environment

The server program has been used on hp9000s700 machines running a `SYSV` compliant operating system (compiled with `-D SYSV5`) and Intel Pentium machines running `LINUX` (compiled with `-D LINUX`). For testing with TACO the server could have been chosen among any of the machines provided by

DIKU that can be reached on the Internet or any of the Intel Pentium machines on the AMIGOS/Net. For convenience[1] the machine chosen during testing with TACO was the Intel Pentium machine `amigos1.diku.dk` running `LINUX`, which also acted as Mobility Support Gateway (see [11]).

The client program without TACO has been used on hp900s700 machines running a `SYSV` compliant operating system and on an Intel Pentium machine running `LINUX`. The machine chosen for testing with TACO was the `amigos6.diku.dk` which is an Intel Pentium notebook running `LINUX` and equiped with a PC Card that fits the requirements.

6.1.2 Portability

$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ standalone could easily ported to other UNIX look-a-likes (I believe), but I do not think it will be easy to port it to other operating systems, since the UNIX file primitives are used extensively. If $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ is to be used with TACO, then it will be a matter of porting TACO as well, and I cannot tell you if that is hard (or even possible) or easy.

6.2 Fault-Tolerance

Since all caching information (on the client side) and all locking information (on the server side) is kept in external files, then none of it is lost if either of the machines should crash! The only thing lost are information to match open file pointers with file names and lock expiration times; but that is hardly a problem, since the locks should (eventually) expire, and a crashed client program is not able to use the file pointers any longer anyway.

Both server program and client applications can be restarted after a crash; the caching and locking info will still be available.

[1] **And** based on advice from Jørgen Sværke Hansen, co-implementor and current maintainer of the TACO implementation

6.3 Client/Server Communication

The server and the client communicates by sending messages of the type `pmessage` defined as:

```
typedef struct pmessage_struct {
  int status;              /* status */
  int request;             /* request or reply */
  long t;                  /* time */
  long ret;                /* readlock expiration time */
  long wet;                /* writelock expiration time */
  long ct;                 /* consistency time */
  long cct;                /* consistency check time */
  long mt;                 /* modification time */
  long crt;                /* client receive time */
  char name[PATHNAME_MAX]; /* filename */
} pmessage;

pmessage buf;
```

The server knows (handles) the following requests in `buf.request`: `READ_OPEN`, `WRITE_OPEN`, `SEND_FILE`, `RECV_FILE`, `READ_LOCK`, `WRITE_LOCK`, `READ_UNLOCK`, `WRITE_UNLOCK`, `TELL_TIME`, `REMOVE_FILE`, `RENAME_FILE`, `READ_CLOSE`, `WRITE_CLOSE`, and `SEND_STAT` defined in `"pesto.h"`.

For all the above requests, the server replies with a status in `buf.status` of `NOT_OK` if something went wrong or `OK` otherwise. If the status is `OK` then the reply is found in `buf.request` and will be one of the following: `CONSISTENT`, `INCONSISTENT`, `NOTFOUND`, `NOTLOCKED`, `ISLOCKED`, `ISUNLOCKED`, `WASLOCKED`, `WASUNLOCKED`, `WASREMOVED`, `WASRENAMED`, `DOSEND`, `WILLSEND`, or `WASFOUND` defined in `"pesto.h"`.

Each request is answered with a reply. If the reply is `DOSEND` or `WILLSEND` then a file is subsequentially send from the client to the server or vice versa.

6.3.1 Communication with TACO

Explicit use of TACO was narrowed down to link monitoring using the link daemon, `linkd`, communicating with the client applications through a newly written pesto-taco daemon, `ptacod`, and the file `".pesto.commstat"`, see Figures 6.1 (p. 77), 6.2 (p. 78), and 6.3 (p. 79). The pesto-taco daemon is based on code from the TACO user guide [12].

Figure 6.1: PeStO-TACO daemon (1)

```
/*****************************************************************************/
/* ptacod.c - PeStO TACO link monitoring Daemon                               */
/*                                                                           */
/* Written by: Michael G. Sørensen, November 1996, DIKU                      */
/*****************************************************************************/

/*
   Based on code from

   "Users Guide for TACO" by Jørgen Sværke Hansen, November 15, 1996
*/

#include "pesto.h"
#include <malloc.h>
#include <mobsup.h>
#include <limits.h>

/* BEGIN define ******************************************************************/
                                /* settings in lnkchg.c */
#define FULL_COST_MAX           10 /*   ether =        0   */
#define FULL_BANDWIDTH_MIN 1000000 /*   ether = 10000000   */
#define FULL_LATENCY_MAX       800 /*   ether =      400   */

#define WEAK_COST_MAX          800 /*   gsm   =      400   */
#define WEAK_BANDWIDTH_MIN    1200 /*   gsm   =     9600   */
#define WEAK_LATENCY_MAX       280 /*   gsm   =      140   */
/* END define ********************************************************************/

int ln_sfd=0;

void io_sa_handler(sig_no)
  int sig_no;
{
  struct qos_if cur_qos;
  int link_up;
  char commstr[17];
  int fd;

  accept_link_notification(&cur_qos);

  link_up=(cur_qos.flags&MIF_UP);

  if(cur_qos.min_bandwidth>=FULL_BANDWIDTH_MIN&&
     cur_qos.max_cost<=FULL_COST_MAX&&
     cur_qos.max_latency<=FULL_LATENCY_MAX) {
    if(link_up) /* CONNECTED */
      sprintf(commstr,"CONNECTED       ");
    else /* DISCONNECTED */
      sprintf(commstr,"DISCONNECTED    ");
  }
  else {
    if(cur_qos.min_bandwidth>=WEAK_BANDWIDTH_MIN&&
       cur_qos.max_cost<=WEAK_COST_MAX&&
       cur_qos.max_latency<=WEAK_LATENCY_MAX) /* WEAKLY CONNECTED */
      sprintf(commstr,"WEAKLY_CONNECTED");
    else /* DISCONNECTED */
      sprintf(commstr,"DISCONNECTED    ");
  }

#ifdef TEST
  printf("%s\n",commstr);
#endif
```

Figure 6.2: PeStO-TACO daemon (2)

```
  if((fd=open(".pesto.commstat",O_WRONLY|O_CREAT,PERMS))!=-1) {
    if(lock_file(fd)!=NOT_OK) {
      if(lseek(fd,0L,0)!=-1)
        write(fd,commstr,strlen(commstr));
      unlock_file(fd);
    }
    close(fd);
  }
}

void setup_sio_signal()
{
  struct sigaction *sa;

  sa=(struct sigaction *)malloc(sizeof(struct sigaction));
  sa->sa_handler=&io_sa_handler;
  sigemptyset(&(sa->sa_mask));
  sa->sa_flags=0;

  sigaction(SIGIO,sa,NULL);

  return;
}

void main()
{
  struct qos_if iface_qos;

  iface_qos.max_cost=INT_MAX;
  iface_qos.min_bandwidth=0;
  iface_qos.max_latency=INT_MAX;
  iface_qos.flags=MIF_UP;

  ln_sfd=assign_link_notification(&iface_qos);

  setup_sio_signal();

  do {
    pause();
  } while(TRUE);
}
```

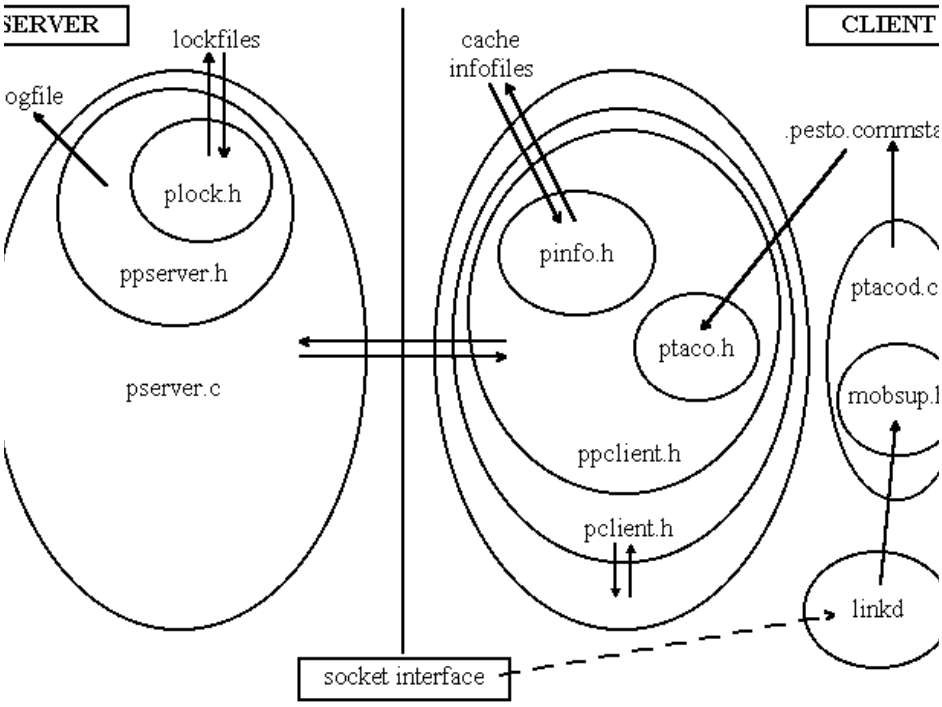
Figure 6.3: PeStO-TACO communication

```
/*****************************************************************************/
/* ptaco.h - include file for PeStO file server programs                     */
/*                                                                           */
/* Written by: Michael G. Sørensen, September-November 1996, DIKU            */
/*****************************************************************************/

int ptaco()
{
  int fd;
  char c;

  /* ptacod writes communication status to .pesto.commstat */

  if((fd=open(".pesto.commstat",O_RDWR))==-1)
    return DISCONNECTED;
  else
    if(lock_file(fd)==NOT_OK) {
      close(fd);
      return DISCONNECTED;
    }
    else {
      if(read(fd,&c,1)<1)  {
        unlock_file(fd);
        return DISCONNECTED;
      }
      unlock_file(fd);
      switch(c) {
        case 'C':
          return CONNECTED;
        case 'W':
          return WEAKLY_CONNECTED;
        default:
          return DISCONNECTED;
      }
    }
}
```

Figure 6.4 (p. 80) shows the data flow. The arrow from the socket interface to the `linkd` daemon should not be taken too literally - it is more complicated than so, see [11]. An overview of the files are given in Section 6.4.

The system naturally uses TACO implicitly when it (TACO) is running, since the TACO system handles the forwarding of all IP and UDP packets to and from the client when weakly connected, see [11].

Figure 6.4: Data flow



6.4 Overview of Files and Subroutines

Here follows a list of the program files and their contents. The list is given here only to provide a minimal overview of the code written. Excerpts from the server program, `pserver.c`, and the include file for client applications, `pclient.h`, are given in Appendix A. An overview of which files are included in which files can be seen in Figure 6.5.

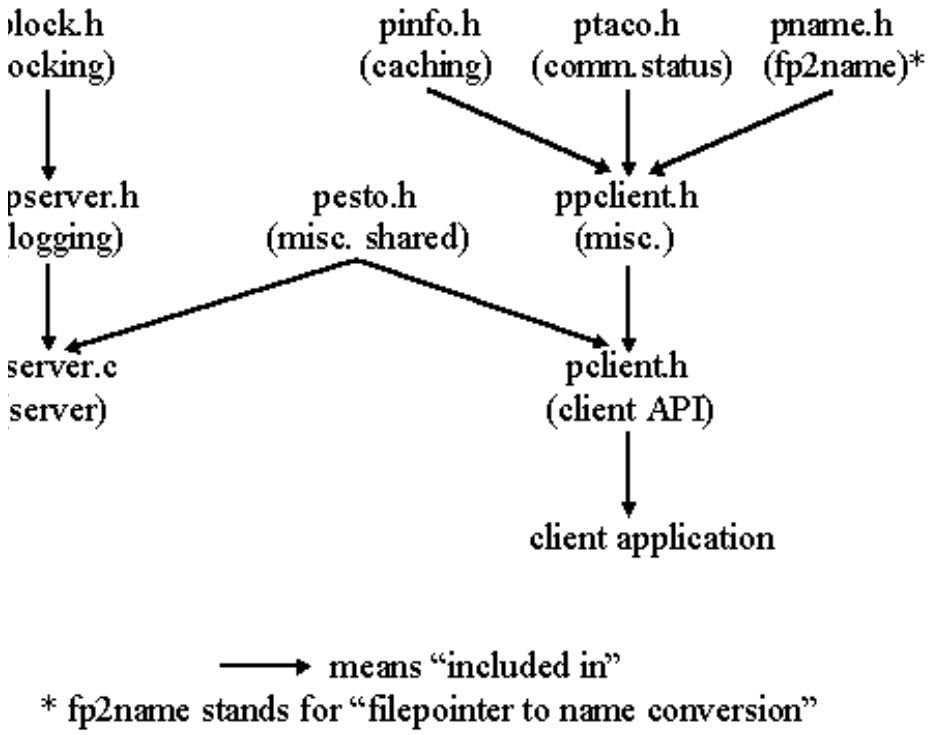
`pclient.h`: Include file for client applications.[2]

- `#include`: `"pesto.h"` & `"ppclient.h"`

- contains all of those primitives described in Section 5.7.1 and Section 5.7.2 except `p_perror`.

[2]Will some day be implemented as a library.

Figure 6.5: Include files



`ptacod.c`: Communication between $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ and TACO. It is in this file that the intervals for connected, weakly connected, and disconnected operation are defined (see Figure 6.1, p. 77).

- `#include`: `"pesto.h"` & `"mobsup.h"`
- `#define`: `FULL_COST_MAX`, `FULL_BANDWIDTH_MIN`, `FULL_LATENCY_MAX`, `WEAK_COST_MAX`, `WEAK_BANDWIDTH_MIN` & `WEAK_LATENCY_MAX`.

Should be linked with the TACO `mob` library:

```
cc -o ptacod -I/home/projects/tacoo/tacosys/moblib/include  \
-L/home/projects/tacoo/tacosys/moblib/lib -D LINUX ptacod.c \
-lmob
```

Is supposed to run on client machine in the background, **and** the TACO link daemon, `linkd`, should also be running.

—

`pserver.c`: $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ server program.

- `#include`: `"pesto.h"` & `"ppserver.h"`
- `void main(int argc,char *argv[])`
- `void p_server()`

Is supposed to be running on the server machine!

—

`pesto.h`: Shared include file for clients and server.

- `p_perror` (see 5.7.2)
- `int init_host(char *host,int port,struct sockaddr_in *sa)`
- `int init_sock(struct sockaddr_in *sa,int *sd)`
- `int send_file(int sd,char *name)`
- `int recv_file(int sd,char *name,mode_t perms)`
- `int close_sock(int sd)`
- `int lock_file(int fd)`
- `int unlock_file(int fd)`

Also contains all necessary includes of standard libraries, all commonly used defines, a type definition of the buffer used for exchanging messages between server and client, and global variables (such as the $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ error number variable `int pso_errno`).

—

`ppclient.h`: Include file for client code. Contains miscellaneous help functions used by the various library functions in `pclient.h`.

- `#include`: `"pinfo.h"`, `"ptaco.h"` & `"pname.h"`
- `typedef`: `NAME` & `_nameb`

- `void pbuf(pmessage *buf,int request,long ret,long wet,long ct,long cct,long mt,char *name)`
- `FILE *pfopen(char *name,char *mode,int fd)`
- `int prequest(struct sockaddr_in *sa,int *sd,pmessage *buf)`
- `int preceive(struct sockaddr_in *sa,int *sd,pmessage *buf)`

—

`pinfo.h`: Include file for client code. Contains functions that read, write, and update caching information for files.

- `void name_info(char *name,char *infoname)`
- `int write_info(int fd,long mt,long ct,long cct,long ret,long wet,long crt)`
- `int read_info(char *name,long *mt,long *ct,long *cct,long *ret,long *wet,long *crt,int *fd)`
- `int update_info(int fd,long mt,long ct,long cct,long ret,long wet,long crt)`
- `int delete_info(char *name)`

—

`ptaco.h`: Include file for client code. Contains the function `int ptaco()` used for determining communication status as reported by `ptacod.c`.

—

`pname.h`: Include file for client code. Contains functions for mapping file pointer to file names and lock expiration times. Uses the `NAME` & `_nameb` structures defined in `ppclient.h` in a similar manner to the handling of open file pointers in [21, Ch.8].

- `int pgetname(FILE *fp,char *name)`
- `int pinsname(FILE *fp,char *name)`

- `int pdelname(FILE *fp)`

—

`ppserver.h`: Include file for server code. Contains miscellaneous help functions used in the server program.

- `#include`: `"plock.h"`
- `#define`: `LOG` & `LOGFILE`
- `void plog(char *str)`
- `void plogerr(char *str,int sd,char *name)`

—

`plock.h`: Include file for server code. Contains functions that read, write, and update locking information for files.

- `name_lock(char *name,char *lockname)`
- `int write_lock(int fd,char *host,long ret,long wet)`
- `int read_lock(char *name,char *host,long *ret,long *wet,int *fd)`
- `int update_lock(int fd,char *host,int ret,int wet)`
- `int lookup_lock(int fd,char *host,long ret,long wet)`
- `int delete_lock(char *name)`

6.5 Program Flow

To give some idea of, what actually goes on within the client library functions, I have drawn flow diagrams for `p_open`, `p_close`, `p_lock`, and `p_unlock`. These are attached (being the makings of a non-LaTeX2ϵ-compatible program) to Appendix C. An explanation is included there.

6.6 Availability

All of the source code is available on request, contact me, and I will be more than willing to provide you with a copy.

Chapter 7

Test & Evaluation

If the file system is to have any success, then the overhead introduced by its implementation on top of TACO and the existing UNIX system should be negligible. I have made a series of tests to measure the overhead (in time).

I have not performed any thorough black-box or internal test, but I do think that the performance tests cover most cases: strict, optimistic, and pessimistic read and write operations, fully connected, weakly connected, and disconnected mode, cached and not cached!

The performance tests are described in Section 7.1, the results are listed in Section 7.2, and an evaluation is made in Section 7.3. The evaluation section also lists problems encountered with using $\mathcal{PESTO}$ and TACO.

7.1 Tests

The test environment for use of TACO has already been described, see Section 6.1.1. Weakly connected mode were tested using a 14400B modem. Another environment used in order to evaluate $\mathcal{PESTO}$ without use of TACO has been with two hp9000s700 machines in DIKU's net; `embla.diku.dk` as server and `skade.diku.dk` as client. In these environments I have performed the following tests (on the files listed in Table 7.1):

1. Session: Reading - fully connected.
   (a) Start with an empty cache.
   (b) Communications status is fully connected.
   (c) Open all the files for reading strict and close them.
   The files are not cached. Files are fetched from the server. This measures the overhead compared with TACO for fetching when fully connected, and compared to remote NFS access.

(d) Open all the files for reading strict and close them.
The files are cached and consistent. Files are acknowledged from the server. This measures the gain of caching compared with TACO for fetching when fully connected.

(e) Communication status is disconnected.

(f) Open all the files for reading optimistic and close them.
The files are cached and within the consistency time bound. Files are opened locally, and there is no server contact. This measures the overhead compared with local UNIX for reading.

(g) Communication status is fully connected.

(h) Open all the files for reading pessimistic and close them.
The files are cached and consistent. Files are acknowledged from and locked on the server. This measure the overhead of locking compared with not locking (in one of the above tests) when fully connected.

2. Session: Reading - weakly connected.

(a) Start with an empty cache.

(b) Communications status is weakly connected.

(c) Open all the files for reading strict and close them.
The files are not cached. Files are fetched from the server. This measures the overhead compared with TACO for fetching when weakly connected.

(d) Open all the files for reading strict and close them.
The files are cached and consistent. Files are acknowledged from the server. This measure the gain of caching compared with TACO for fetching when weakly connected.

(e) Open all the files for reading optimistic and close them.
The files are cached and within the consistency time bound. File are opened locally, ant here is no server contact. This measures the overhead compared with local UNIX for reading.

(f) Open all the files for reading pessimistic and close them.
The files are cached and consistent. Files are acknowledged from and locked on the server. This measures the overhead of locking compared with no locking (in one of the above tests) when weakly connected.

3. Session: Writing - fully connected.

(a) Start with an empty cache.

(b) Communication status is fully connected.

(c) Open all the files for writing strict, write to them, and close them. The files are not cached, and they are updated. Files are fetched from and send to the server. This measures the overhead compared with TACO for fetching and sending when fully connected, and compared to remote NFS access.

(d) Communication status is disconnected.

(e) Open all the files for writing optimistic, write to them, and close them.
The files are cached and within the modification time bound, and they are updated. Files are opened locally, and send to the server, later. This measures the overhead compared with ANSI-C on top of UNIX for writing.

(f) Communication status is fully connected.

(g) Open all the files for writing optimistic, write to them, and close them.
The files are cached and within the modification time bound, and they are updated. Files are opened locally, and send to the server. This measures the overhead compared with TACO for sending when fully connected.

4. Session: Writing - weakly connected.

(a) Start with an empty cache.

(b) Communication status is weakly connected.

(c) Open all the files for writing strict, write to them, and close them. The files are not cached, and they are updated. Files are fetched from and send to the server. This measures the overhead compared with TACO for fetching and sending when weakly connected.

(d) Open all the files for writing optimistic, write to them, and close them.
The files are cached and within the modification time bound, and they are updated. Files are opened locally, and send to the server. This measures the overhead compared with TACO for sending when weakly connected.

Table 7.1: Files used for testing

No.	File	Size	Type
1	chb.ps	63499	Postscript
2	ctb2.ps	24258	Postscript
3	dataflow.ps	83551	Postscript
4	etb.ps	14812	Postscript
5	files2.ps	38401	Postscript
6	included.ps	49241	Postscript
7	mtb2.ps	24258	Postscript
8	rwc-isde.ps	91006	Postscript
9	rwc-none.ps	83218	Postscript
10	rwc-unde.ps	83218	Postscript
11	so2pesto.ps	83881	Postscript
12	states2.ps	57876	Postscript
13	speciale.log	9890	text
14	speciale.idx	10597	text
15	speciale.aux	20467	text
16	speciale.dvi	278036	DVI
17	speciale.toc	7454	text
18	speciale.lof	1625	text
19	speciale.lot	1675	text
20	speciale.ind	6403	text
21	speciale.ilg	323	text
22	speciale.tex	193688	LaTeX2ϵ
23	pclient.h	41968	C
24	pesto.h	9019	C
25	pinfo.h	3031	C
26	plock.h	4745	C
27	pname.h	1525	C
28	ppclient.h	3873	C
29	ppserver.h	1113	C
30	pserver.c	21376	C
31	ptaco.h	977	C
32	ptacod.c	2730	C
		1317734	

7.2 Results

The running times for the tests are listed in Figures 7.2 and 7.3. The times are averages for at least two test runs. The $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO and TACO tests were performed using the AMIGOS machines and the $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$, NFS, and UNIX tests were performed using the hp9000s700 machines (see previous section).

Table 7.2: Running times (in seconds) for reading

NO.	TEST	SYSTEM	TIME
1c	Not cached, fetch, fully connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	5
		TACO	2
		$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$	10
		NFS	<1
1d	Cached, acknowledge, fully connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	1
		$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$	5
1f	Cached, disconnected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	<1
		TACO	<1
		$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$	1
		UNIX	<1
1h	Cached, acknowledge & lock, fully connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	2
		$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$	6
2c	Not cached, fetch, weakly connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	1021
		TACO	982
2d	Cached, acknowledge, weakly connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	50
2e	Cached, weakly connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	25
2f	Cached, acknowledge & lock, weakly connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	49

It may come as a surprise that some of the running times on the relatively small AMIGOS machines are less than those from the big hp9000s700 machines. It should probably be contributed to the fact that during testing I had the AMIGOS machines (and the full network bandwidth) for myself whereas the hp9000s700 machines (and the network bandwidth) were shared with up to 30-40 (eager) undergraduate students!

Table 7.3: Running times (in seconds) for writing

NO.	TEST	SYSTEM	TIME
3c	Not cached, fetch and send, fully connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	6
		TACO	5
		$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$	15
		NFS	2
3e	Cached, disconnected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	<1
		TACO	<1
		$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$	1
		UNIX	<1
3g	Cached, send, fully connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	6
		TACO	2
		$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$	16
4c	Not cached, fetch and send, weakly connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	2061
		TACO	2009
4d	Cached, send, weakly connected	$\mathcal{P}^e\mathcal{S}^t\mathcal{O}$+TACO	1033
		TACO	1003

7.3 Evaluation

ERRATUM HUMANUM EST

These are the conclusion that can be drawn from the running times:

- If the cached files are used then the gain (compared to no caching) is approximately 95% (e.g., 2d (50) compared to 2c (982)) when weakly connected, and approximately 50% when fully connected (e.g., 1c (2) compared to 1d (1)).
- Disconnected operation has no overhead (1f, 3e).
- The overhead when weakly connected is negligible (2c,4c, and 4d).
- The overhead when fully connected is acceptable (1c, 3c, and 3g)—although in percentage rather high, then it must be remembered that the overhead only occurs on open and close, and shared files should have a lifetime that make this overhead negligible (i.e., short-lived temporary files should not use $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$).

- Locking requires virtually no overhead (1d compared to 1h, and 2d compared to 2f).

All in all, I find these results to be acceptable. The overhead when fully connected may need some improvement.

7.3.1 Problems with PeStO

I have encountered the following problems with $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$:

1. On the server empty lockfiles are created. This is really not a problem, but it is does not look "pretty".

2. Filenames must have the same meaning on the server and on the client. Thus if the client application references opens a file `"foo.txt"` then the primary copy of `"foo.txt"` must be in the same directory as the server program was started from, i.e., current working directory. During testing this of course posed no problem, but if the system is to be used for real someday, then an alternative solution may have to be considered.

3. When disconnected and using close *ETB* of zero,[1] then child processes awaiting better communication are created. If this is done many times then the (UNIX) system may run out space for additional open file pointers and/or socket descriptors, or even result in too many processes.

4. If client and server are **not** synchronized then $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ **will** behave unexpectedly. The current version of the server actually accepts requests initiated by the client **after** the server has received it!

I believe that the first and second problem are easy to live with. I do not know about the third one! A way to solve that one, could be to let a single process (e.g., a daemon) handle all "pending" closes. All in all I think that $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ works OK. The fourth problem was expected and is solved by remembering to keep the clients synchronized with the server—it is easy to do, but it **must** be remembered!

[1] Or any *ETB* that times out before any type of connection is made.

7.3.2 Problems with TACO

I have encountered the following problems when using TACO:

1. The communication between the link daemon, `linkd`, and the pesto-taco daemon, `ptacod`, did not work properly. Entering and leaving *weakly connected* operation were not acknowledged. During testing the entering and leaving weakly connected operation were done using:

   ```
   echo WEAKLY CONNECTED >.pesto.commstat
   echo DISCONNECTED     >.pesto.commstat
   ```

   Tests using a simple link changing simulation program showed that the fault was **not** within my program.

2. The client often halted when re-entering *connected* operation (i.e., re-establishing the Ethernet connection) after (voluntary) disconnections. In these cases it was necessary to re-boot the machines in order to get a full connection. This was quite frustrating!

Chapter 8

Conclusions

8.1 Contributions

The two main contributions of this report are:

- A new distributed file system with support for mobile computing.
- Guidelines for implementing a transactional facility there upon.

The fulfillment of the goals for the distributed file system is discussed in Section 8.2.

As for the guidelines for a transactional facility—I have

- given my ideas to which properties the transactions should have,
- specified a new algorithm for optimistic concurrency control,
- specified transaction primitives, and
- given a rich example of how I foresee the transactions used.

8.2 Fulfillment of Goals

"Pretty still no star
want to go so far"
– Dizzy Mizz Lizzy (Run)

I think that the three main design goals, see Section 1.5.1 (p. 9), have been fulfilled:

1. Applications are able to utilize any desired level of optimism or pessimism,

2. applications are able to adapt their behaviour according to different communication characteristics, and

3. the porting of existing applications should be fairly easy.

I believe that the first goal is fulfilled completely. The fulfillment of the second goal has a minor flaw; the adaptation is only for three rather coarse grained characteristics of communication (connected, weakly connected, and disconnected).[1] The fulfillment of the third goal can be criticized for the fact that it has not been tried—but a scheme for how to do it, rudimentarily, is given in Section 5.8.

The overhead introduced by the system has shown to be acceptable, see Section 7.2, especially due to the fact that the small, but noticeable overhead that **is** there only occurs when opening and closing files—reads and writes have no overhead (they are in fact the same).

8.3 Future Work

An endless amount of future work can be suggested, but to name a few:

- The environment used in this thesis has been very simple and could easily be broadened or generalized, e.g., multiple servers, peer-to-peer communications, etc. There are many open roads.

- I think that it will be possible to have greater (better) utilization of TACO's QoS facilities. Maybe it would be a good idea to let applications specify both time bound and quality of service parameters. Better integration of $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ and TACO is a certainly worth looking at.

- Other consistency measures than time might be considered; the number of updates to a file, the difference (in terms of contents) between a replica and the primary copy, ... Notions of *fidelity* as in the Odyssey [43] system might also be included in the model.

- An implementation of the suggested transactional facility, i.e., AMIGOS phase three.

[1] There is, however, no evidence to support the need for finer grained adaptation; existing mobile computing systems, such as Coda and Little Work, uses these three modes of operation.

- An investigation of whether the ideas from this thesis make sense in an object-oriented environment, i.e., AMIGOS phase four and five.

8.4 Conclusion

"That's all Folks!"
– Looney Tunes

I have designed and implemented a distributed file system with support for mobile computing. It enables applications to utilize any level of optimism or pessimism and to adapt their behaviour according to different modes of communication. The implementation have shown the design to be feasible, but since no existing application has been ported to and used with the system, the viability is a somewhat open question. I believe it is viable.

I have layed down guidelines for the implementation of a transactional facility on top of the file system. I think, I have captured most of the needed considerations in my doing so.

I must admit that I at times regretted the initial decision to use the TACO system—it was unavailable for some time,[2] and its behaviour was not always predictable. Still, thanks to hard work (and some help), results were made in the end.

I think that the TACO system and $\mathcal{P}^e\mathcal{S}^t\mathcal{O}$ show promising results—but they (by themselves and together) probably have some way to go, before reaching "the top".

8.5 Postscriptum

Well, I am through with school, and I am heading of into "the real world". Before I go, I have to tell you...

John Lennon has stated in one of his songs that:

"Life is what happens to you
while your busy making other plans."

[2]Due to the changing of IP-numbers shortly after I had started my testing.

Figure 8.1: An alternative to Mobile Computing

After adjusting the ergonomic design of his home office, Dave's job-related stress was reduced by 70%.

During the writing of this report, my girl has agreed to marry me, and we have learned that we are to be parents (soon):

Christina,
I dedicate this work to you

Appendix A

Program

A.1 Server: pserver.c

```
/*****************************************************************************/
/* pserver.c - server process for PeStO file server                          */
/*                                                                           */
/* Written by: Michael G. Sørensen, September-November 1996, DIKU            */
/*****************************************************************************/

/* BEGIN include *************************************************************/
#include "pesto.h"    /* include file for PeStO client and PeStO server */
#include "ppserver.h" /* include file for PeStO server  */
/* END include ***************************************************************/

/* BEGIN globals *************************************************************/
int sd1,sd2;                /* socket descriptors */
struct sockaddr_in sa1,sa2; /* socket addresses (on the Internet) */
long t;
struct linger l;
/* END globals ***************************************************************/

/* BEGIN PeStO file server ***************************************************/
void p_server()
{
  char host[HOSTNAME_MAX];          /* name of host */
  struct hostent *hp;               /* result of hostname lookup */
  pmessage buf;                     /* message buffer */
  struct stat stbuf;                /* file status buffer */
  int locked,readlocked,writelocked;
  long ret,wet,rlet,wlet;
  int fd;
  int flag;
  char lockname[LOCKNAME_MAX];
  char logstr[1000];

  char *inet_ntoa();

  close_sock(sd1);

  if((hp=gethostbyaddr((char *)&sa2.sin_addr,sizeof(struct in_addr),sa2.sin_family))==NULL)
    strcpy(host,inet_ntoa(sa2.sin_addr));
  else
```

```
    strcpy(host,hp->h_name);

#ifdef LOG
  time(&t);
  sprintf(logstr,"Startup from %s port %u at %s",host,ntohs(sa2.sin_port),ctime(&t));
  plog(logstr);
#endif

  l.l_onoff=1;
  l.l_linger=1;
  if(setsockopt(sd2,SOL_SOCKET,SO_LINGER,&l,sizeof(struct linger))==-1) {
    pso_errno=PSO_EERROR;
    close_sock(sd2);
    exit(NOT_OK);
  }

  if(recv(sd2,&buf,sizeof(buf),0)!=sizeof(buf)) {
    pso_errno=PSO_EERROR;
    close_sock(sd2);
    exit(NOT_OK);
  }

#ifdef LOG
  sprintf(logstr,"Request from client initiated %s",ctime(&(buf.t)));
  plog(logstr);
#endif

  /* handle the request and give a reply */

  switch(buf.request){
    case READ_OPEN:
    case WRITE_OPEN:
      /* BEGIN open for READING or WRITING ************************************/

      /*
         Possible returns are: buf.status is set to NOT_OK or OK, and if OK then
         buf.request is set to CONSISTENT, INCONSISTENT, NOTFOUND, ISLOCKED or
         WASLOCKED.

         If CONSISTENT then buf.ct and buf.cct are set. If INCONSISTENT then
         buf.mt and buf.cct are set. If ISLOCKED or WASLOCKED then also
         CONSISTENT if buf.ct is equal to buf.ct else INCONSISTENT.

         If buf.request is READ_OPEN then the content of buf.wet is ignored,
         and similarly the content of buf.ret is ignored if buf.request is
         WRITE_OPEN.
      */

#ifdef LOG
      if(buf.request==READ_OPEN)
        sprintf(logstr,"READ_OPEN %s\n",buf.name);
      else
        sprintf(logstr,"WRITE_OPEN %s\n",buf.name);
      plog(logstr);
#endif

      buf.status=OK;

      if(stat(buf.name,&stbuf)==-1) /* check existense and status of file */ {
        if(errno==ENOENT) /* No such file or directory */
          buf.request=NOTFOUND;
        else
```

```
    buf.status=NOT_OK;
}
else {
  if((locked=read_lock(buf.name,host,&ret,&wet,&fd))==NOT_OK)
    buf.status=NOT_OK;
  else {
    time(&t);
    readlocked=(locked&&ret>=t);
    writelocked=(locked&&wet>=t);

    buf.cct=t; /* set consistency check time for file */
    if(stbuf.st_mtime==buf.mt) /* check modification time of file */ {
      flag=CONSISTENT;
      buf.ct=t;
    }
    else {
      flag=INCONSISTENT;
      buf.mt=stbuf.st_mtime;
    }

    if(writelocked||(readlocked&&buf.request==WRITE_OPEN))
      buf.request=ISLOCKED;
    else {
      if((buf.request==READ_OPEN&&buf.ret<t)||
         (buf.request==WRITE_OPEN&&buf.wet<t)) /* no locking required */ {
        if((buf.request==READ_OPEN&&buf.ret>0L)||
           (buf.request==WRITE_OPEN&&buf.wet>0L))
          buf.request=NOTLOCKED;
        else
          buf.request=flag;
      }
      else /* locking required */ {
        /*
           At this point we know that the client wishes to put a readlock
           on the file. But that will only be the case if neither a read-
           nor a writelock was previously obtained. Still we might have an
           out-dated (timeout'ed) entry in the lock(file).
        */
        if((locked=lookup_lock(fd,host,&ret,&wet))==NOT_OK)
            buf.status=NOT_OK;
        else {
          if(locked&&(wet>=t||ret>=t))
            /*
               Note, that this should not be! Here we have a client asking for
               a lock although the client already has one. This could be the
               result of unsynchronized clocks.
            */
            buf.status=NOT_OK;
          else {
            if(buf.request==READ_OPEN) {
              rlet=buf.ret;
              wlet=0L;
            }
            else {
              wlet=buf.wet;
              rlet=0L;
            }
            if(locked) {
              if(update_lock(fd,host,rlet,wlet)==NOT_OK)
                buf.status=NOT_OK;
              else
                buf.request=WASLOCKED;
```

```
                }
                else {
                  if(write_lock(fd,host,rlet,wlet)==NOT_OK)
                    buf.status=NOT_OK;
                  else
                    buf.request=WASLOCKED;
                }
              }
            }
          }

          if(unlock_file(fd)==NOT_OK)
            buf.status=NOT_OK;
        }
      }

      psend(sd2,&buf);
      break;
      /* END open for READING or WRITING ************************************/

    /* ... */

    default:
      /* BEGIN request UNKNOWN **********************************************/

      /*
         Returns buf.status set to NOT_OK.
      */

#ifdef LOG
      sprintf(logstr,"UNKNOWN\n");
      plog(logstr);
#endif

      buf.status=NOT_OK;
      psend(sd2,&buf);
      break;
      /* END request UNKNOWN ************************************************/
  }

#ifdef LOG
  time(&t);
  sprintf(logstr,"Completed %s port %u at %s",host,ntohs(sa2.sin_port),ctime(&t));
  plog(logstr);
#endif

  close_sock(sd2);
}
/* END PeStO file server ****************************************************/

/* BEGIN main ***************************************************************/
void main()
{
  struct hostent *hp;           /* result of hostname lookup */
  int len;                      /* address length */
  char localhost[HOSTNAME_MAX];

#ifndef SYSV5
#ifndef LINUX
  int fd;
  int sig_child();
```

```
#endif
#endif

  memset((char *)&sa1,0,sizeof(struct sockaddr_in));
  memset((char *)&sa2,0,sizeof(struct sockaddr_in));

  if(gethostname(localhost,HOSTNAME_MAX)==-1) {
    pso_errno=PSO_EERROR;
    exit(NOT_OK);
  }

  if((hp=gethostbyname(localhost))==NULL) {
    pso_errno=PSO_EERROR;
    exit(NOT_OK);
  }

  if((sa1.sin_family=hp->h_addrtype)!=AF_INET) {
    pso_errno=PSO_EBADADDRESS;
    exit(NOT_OK);
  }

  sa1.sin_port=ntohs(PORT);

  if((sd1=socket(AF_INET,SOCK_STREAM,0))==-1) {
    pso_errno=PSO_EERROR;
    exit(NOT_OK);
  }

  if(bind(sd1,(void *)&sa1,sizeof(struct sockaddr_in))==-1) {
    pso_errno=PSO_EERROR;
    close_sock(sd1);
    exit(NOT_OK);
  }

  if(listen(sd1,10)==-1) {
    pso_errno=PSO_EERROR;
    close_sock(sd1);
    exit(NOT_OK);
  }

#ifdef SYSV5
  setpgrp();
#else
#ifndef LINUX
  setpgrp(0,getpid());
  if((fd=open("/dev/tty",O_RDWR))>=0) {
    ioctl(fd,TIOCNOTTY,(char *)NULL); /* loose controlling tty */
    close(fd);
  }
#else
  setpgrp();
#endif
#endif

  switch(fork()) {
    case -1:
      pso_errno=PSO_EERROR;
      close_sock(sd1);
      exit(NOT_OK);
    case 0: /* child process */
      fclose(stdin);
      fclose(stderr);
```

```
#ifdef SYSV5
      signal(SIGCLD,SIG_IGN);
#else
#ifndef LINUX
      signal(SIGCLD,sig_child);
#else
      signal(SIGCLD,SIG_IGN);
#endif
#endif
      for(;;) {
        len=sizeof(struct sockaddr_in);
        if((sd2=accept(sd1,(void *)&sa2,&len))==-1) {
          pso_errno=PSO_EERROR;
          close_sock(sd1);
          exit(NOT_OK);
        }

        switch(fork()) {
          case -1:
            pso_errno=PSO_EERROR;
            close_sock(sd1);
            close_sock(sd2);
            exit(NOT_OK);
          case 0: /* child process */
            p_server();
            exit(OK);
          default: /* parent process */
            close_sock(sd2);
        }
      }

    default: /* parent process */
      close_sock(sd1);
      exit(OK);
  }
}
/* END main ***************************************************************/
```

A.2 Client: pclient.h

```
/*****************************************************************************/
/* pclient.h - client process(es) for PeStO file server                      */
/*                                                                           */
/* Written by: Michael G. Sørensen, September-November 1996, DIKU            */
/*****************************************************************************/

/* BEGIN include ************************************************************/
#include "pesto.h"    /* include file for PeStO client and PeStO server */
#include "ppclient.h" /* include file PeStO client */
/* END include **************************************************************/

/* BEGIN library subroutines ************************************************/
FILE *p_open(name,mode,tb)
  char *name; /* pathname */
  char *mode; /* filemode */
  int tb;     /* time bound (in minutes) */
{
  int sd;                     /* socket descriptor */
  pmessage buf;               /* message buffer */
```

```
long mt,ct,cct,ret,wet,crt;
long ctb,mtb,t,rlet,wlet;
int fd;
int cached,cf,commstat,rewrite,within;

pso_errno=PSO_ENOERROR;

if(strchr(mode,'w')!=NULL||strchr(mode,'a')!=NULL||
  (strchr(mode,'r')!=NULL&&strchr(mode,'+')!=NULL)) {

  /* BEGIN open for WRITING ************************************************/

  /* ... */

  /* END open for WRITING **************************************************/
}
else {
  if(strchr(mode,'r')!=NULL) {

  /* BEGIN open for READING ************************************************/

    if((cached=read_info(name,&mt,&ct,&cct,&ret,&wet,&crt,&fd))==NOT_OK)
      return NULL;

    time(&t);
    ctb=60*(long)tb;  /* consistency time bound (in seconds) */

    if(!cached) {

      /* BEGIN open for READING and NOT CACHED *******************************/

      /* ... */

      /* END open for READING and NOT CACHED *********************************/
    }
    else {

     /* BEGIN open for READING and CACHED ***********************************/

      cf=(ct==ctb);

      if(ctb<0) {

        /* BEGIN open for READING and CACHED and OPTIMISTIC *****************/

        if(wet>=t||ret>=t) /* file is read- or writelocked */
          return pfopen(name,mode,fd,0L,0L);

        if((commstat=ptaco())==NOT_OK) {
          unlock_file(fd);
          return NULL;
        }

        within=(t+ctb<=ct||(wet>0&&t+ctb<=wet)||(ret>0&&t+ctb<=ret));

        if(commstat!=CONNECTED) /* DISCONNECTED or WEAKLY CONNECTED */ {
          if(within)
            /*
               At this point we know that the cached file is within the
               specified (consistency) time bound.

               We do not care whether we know the cached file is consistent
```

```
       (cf) or not (!cf).
    */
    return pfopen(name,mode,fd,0L,0L);
  else {
    pso_errno=PSO_ENOTWITHIN;
    unlock_file(fd);
    return NULL;
  }
}

/* send request to server and receive reply */

pbuf(&buf,READ_OPEN,0L,0L,ct,cct,mt,name);
if(prequest(&sa,&sd,&buf)==NOT_OK||buf.status==NOT_OK) {
  /*
     We could not request the file from the server, so we use the
     cached version (if its within the specified time bound).
  */
  if(within) {
    close_sock(sd);
    return pfopen(name,mode,fd,0L,0L);
  }
  else {
    pso_errno=PSO_ENOTWITHIN;
    unlock_file(fd);
    close_sock(sd);
    return NULL;
  }
}

if(buf.request==ISLOCKED)
  /*
     Note, that even if the file is writelocked on the server we will
     read it - being optimistic.
  */
  if(mt==buf.mt)
    buf.request=CONSISTENT;
  else
    buf.request=INCONSISTENT;

switch(buf.request) {
  case NOTFOUND:
    if(within) {
      pso_errno=PSO_ENOTWITHIN;
      break;
    }
    else {
      close_sock(sd);
      return pfopen(name,mode,fd,0L,0L);
    }

  case INCONSISTENT:
    pbuf(&buf,SEND_FILE,0L,0L,buf.ct,buf.cct,buf.mt,name);
    if(preceive(&sa,&sd,&buf)==NOT_OK) {
      if(within) {
        close_sock(sd);
        return pfopen(name,mode,fd,0L,0L);
      }
      else {
        pso_errno=PSO_ENOTWITHIN;
        break;
      }
```

```
            }
            else
              crt=buf.crt;

        case CONSISTENT:
            if(update_info(fd,buf.mt,buf.ct,buf.cct,ret,wet,crt)==NOT_OK)
              break;
            else {
              close_sock(sd);
              return pfopen(name,mode,fd,0L,0L);
            }

          case NOTLOCKED:
          case WASLOCKED:
            /*
               Note, that the file should not have been attempted locked or
               locked,  because we did not ask for a lock (buf.ret=0L)!
            */
            pso_errno=PSO_ESERVERERROR;
            break;

          default:
            pso_errno=PSO_EUNKNOWNREPLY;
        }

        unlock_file(fd);
        close_sock(sd);
        return NULL;

        /* END open for READING and CACHED and OPTIMISTIC ******************/
      }
      else
        if(ctb==0) {

          /* BEGIN open for READING and CACHED and STRICT ******************/

          /* ... */

          /* END open for READING and CACHED and STRICT ********************/
        }
        else {

          /* BEGIN open for READING and CACHED and PESSIMISTIC **************/

          /* ... */

          /* END open for READING and CACHED and PESSIMISTIC ****************/
        }
      /* END open for READING and CACHED ***********************************/
    }
    /* END open for READING ************************************************/
  }
  else {
    pso_errno=PSO_EBADMODE;
    return NULL;
  }
 }
}

int p_close(fp,etb)
  FILE *fp;
  int etb;  /* expiration time bound (in minutes) */
```

```
{
  char name[PATHNAME_MAX];
  int cached,commstat,request,status,updated;
  long t,et,mt,ct,cct,ret,wet,crt,rlet,wlet;
  int fd,sd;
  struct stat stbuf;
  pmessage buf;

  pso_errno=PSO_ENOERROR;

  time(&t);
  if(etb<0)
    etb=0;
  if(etb>0)
    et=t+60*(long)etb; /* expiration time */

#ifdef SYSV5
  if((fp->_flag&01)==0)
    request=WRITE_CLOSE;
  else
    request=READ_CLOSE;
#else
  if((fp->_flags&04)==0)
    request=READ_CLOSE;
  else
    request=WRITE_CLOSE;
#endif

  if(pgetname(fp,name,&rlet,&wlet)==NOT_OK) {
    pso_errno=PSO_ENOTFOUND;
    fclose(fp);
    return NOT_OK;
  }

  if((cached=read_info(name,&mt,&ct,&cct,&ret,&wet,&crt,&fd))==NOT_OK) {
    pdelname(fp);
    fclose(fp);
    return NOT_OK;
  }

  if(!cached) {
    pso_errno=PSO_ENOTCACHED;
    delete_info(name);
    pdelname(fp);
    fclose(fp);
    unlock_file(fd);
    return NOT_OK;
  }

  if(ct!=cct) {
    pso_errno=PSO_ENOTCONSISTENT;
    pdelname(fp);
    fclose(fp);
    unlock_file(fd);
    return NOT_OK;
  }

  if(stat(name,&stbuf)==-1) {
    if(errno==ENOENT)
      pso_errno=PSO_ENOTFOUND;
    else
      pso_errno=PSO_EERROR;
```

```
    pdelname(fp);
    fclose(fp);
    unlock_file(fd);
    return NOT_OK;
  }

  if(stbuf.st_mtime>=crt&&request==WRITE_CLOSE)
    updated=UPDATED;
  else
    updated=NOT_UPDATED;

  if((commstat=ptaco())==NOT_OK) {
    unlock_file(fd);
    return NOT_OK;
  }

  if(commstat==DISCONNECTED) {
    if(etb==0) {
      if(pdelname(fp)==NOT_OK) {
        fclose(fp);
        unlock_file(fd);
        return NOT_OK;
      }
      if(fclose(fp)==EOF) {
        pso_errno=PSO_ENOTCLOSED;
        unlock_file(fd);
        return NOT_OK;
      }
      if(unlock_file(fd)==NOT_OK)
        return NOT_OK;
    }
    else {
      /* wait for better communication status or timeout on et */
      while((commstat=ptaco())!=DISCONNECTED&&et<t) {
        sleep(SLEEPTIME);
        time(&t);
      }
    }
  }

  if(commstat==DISCONNECTED) {
    if(updated) {
      switch(fork()) {
        case -1:
          pso_errno=PSO_EERROR;
          return NOT_OK;
        case 0: /* child process - wait for better communication */
          fclose(stdin);
          fclose(stderr);
#ifdef SYSV5
          signal(SIGCLD,SIG_IGN);
#else
#ifndef LINUX
          signal(SIGCLD,sig_child);
#else
          signal(SIGCLD,SIG_IGN);
#endif
#endif
          while((commstat=ptaco())!=DISCONNECTED)
            sleep(SLEEPTIME);
          pbuf(&buf,request,rlet,wlet,ct,cct,mt,name);
          pclosesend(&sa,&sd,&buf,fd,updated,crt);
```

```
          exit(0);
        default: /* parent */
          return TIMEOUT;
      }
    }
    else
      return TIMEOUT;
  }

tryagain:

  /* send request to server and receive reply */

  pbuf(&buf,request,rlet,wlet,ct,cct,mt,name);
  if(prequest(&sa,&sd,&buf)==NOT_OK||buf.status==NOT_OK) {
    if(etb==0) {
      if(pdelname(fp)==NOT_OK) {
        fclose(fp);
        unlock_file(fd);
        close_sock(sd);
        return NOT_OK;
      }
      if(fclose(fp)==EOF) {
        pso_errno=PSO_ENOTCLOSED;
        unlock_file(fd);
        close_sock(sd);
        return NOT_OK;
      }
      if(unlock_file(fd)==NOT_OK) {
        close_sock(sd);
        return NOT_OK;
      }
      commstat=DISCONNECTED;
    }
    else {
      while((commstat=ptaco())!=DISCONNECTED&&et<t) {
        sleep(SLEEPTIME);
        time(&t);
      }

      if(commstat!=DISCONNECTED&&et>t) {
        close_sock(sd);
        goto tryagain;
      }
    }
  }

  if(commstat==DISCONNECTED) {
    if(updated) {
      switch(fork()) {
        case -1:
          pso_errno=PSO_EERROR;
          return NOT_OK;
        case 0: /* child process - wait for better communication */
          fclose(stdin);
          fclose(stderr);
#ifdef SYSV5
          signal(SIGCLD,SIG_IGN);
#else
#ifndef LINUX
          signal(SIGCLD,sig_child);
#else
```

```
          signal(SIGCLD,SIG_IGN);
#endif
#endif
          while((commstat=ptaco())!=DISCONNECTED)
            sleep(SLEEPTIME);
          pbuf(&buf,request,rlet,wlet,ct,cct,mt,name);
          pclosesend(&sa,&sd,&buf,fd,updated,crt);
          exit(0);
        default: /* parent */
          close_sock(sd);
          return TIMEOUT;
      }
    }
    else {
      close_sock(sd);
      return TIMEOUT;
    }
  }

  switch(buf.request) {
    case INCONSISTENT: /* has been updated on the server in the mean time */
    case ISLOCKED:     /* has been  locked on the server in the mean time */
      status=FAILURE;
      break;

    case CONSISTENT:
      if(updated) {
        if(pdelname(fp)==NOT_OK) {
          pso_errno=PSO_EERROR;
          fclose(fp);
          break;
        }
        if(fclose(fp)==-1) {
          pso_errno=PSO_EERROR;
          break;
        }
        else {
          pbuf(&buf,RECV_FILE,0L,0L,buf.ct,buf.cct,buf.mt,name);
          if(prequest(&sa,&sd,&buf)==NOT_OK||buf.status==NOT_OK) {
            if(buf.status==NOT_OK)
              pso_errno=PSO_ESERVERERROR;
            break;
          }
          else {
            if(send_file(sd,name)==NOT_OK)
              break;
            else {
              pbuf(&buf,SEND_STAT,0L,0L,buf.ct,buf.cct,buf.mt,name);
              if(prequest(&sa,&sd,&buf)==NOT_OK||buf.status==NOT_OK) {
                if(buf.status==NOT_OK)
                  pso_errno=PSO_ESERVERERROR;
                break;
              }
              else {
                if(wlet>t)
                  wlet=t;
                if(update_info(fd,buf.mt,buf.ct,buf.cct,ret,wlet,crt)==NOT_OK)
                  break;
                else {
                  status=SUCCESS;
                  break;
                }
```

```
            }
          }
        }
      }
    }
    else {
      if(request==READ_CLOSE)
        if(rlet>t)
          rlet=t;
        else
          rlet=ret;
      if(request==WRITE_CLOSE)
        if(wlet>t)
          wlet=t;
        else
          wlet=wet;
      if(update_info(fd,buf.mt,buf.ct,buf.cct,rlet,wlet,crt)==NOT_OK)
        break;
      else {
        status=SUCCESS;
        break;
      }
    }

  default:
    pso_errno=PSO_EUNKNOWNREPLY;
}

if(pso_errno==PSO_ENOERROR) {
  if(!updated) {
    if(pdelname(fp)==NOT_OK) {
      fclose(fp);
      unlock_file(fd);
      close_sock(sd);
      return NOT_OK;
    }
    if(fclose(fp)==EOF) {
      pso_errno=PSO_ENOTCLOSED;
      unlock_file(fd);
      close_sock(sd);
      return NOT_OK;
    }
  }
  if(unlock_file(fd)==NOT_OK) {
    close_sock(sd);
    return NOT_OK;
  }
  close_sock(sd);
  return status;
}
else {
  if(!updated) {
    pdelname(fp);
    fclose(fp);
  }
  unlock_file(fd);
  close_sock(sd);
  return NOT_OK;
}
}

int p_lock(name,mode,letb)
```

```
  char *name,*mode;
  int letb;         /* lock expiration time bound (in minutes) */
{
  /* ... */
}

int p_unlock(name,mode)
  char *name,*mode;
{
  /* ... */
}

int p_remove(name)
  char *name;
{
  /* ... */
}

int p_stat(name,mt,ct,cct,ret,wet)
  char *name;
  long *mt,*ct,*cct,*ret,*wet;
{
  int cached;
  long crt;
  int fd;

  pso_errno=PSO_ENOERROR;

  if((cached=read_info(name,mt,ct,cct,ret,wet,&crt,&fd))==NOT_OK)
    return NOT_OK;
  else {
    if(!cached)
      if(delete_info(name)==NOT_OK)
        return NOT_OK;
    if(unlock_file(fd)==NOT_OK)
      return NOT_OK;
    else
      return cached;
  }
}

int p_comm()
{
  pso_errno=PSO_ENOERROR;

  return ptaco();
}

long p_time(flag)
  int flag;
{
  /* ... */
}
/* END library subroutines ************************************************/
```

Appendix B

Examples

Bank Account

Here is the classical Bank Account example as it is presented by ORACLE in [36, Ch.3, p.27]:

> "A *transaction* is a user-defined series of logically related SQL operations. All changes brought by the SQL operations are either undone or made permanent at the same time.
>
> You perform transactions with the COMMIT or ROLLBACK statements. You use these statements to ensure that either all or none of your changes are made to the database.
>
> A simple example of a transaction involves transferring money from one bank account to another. The transaction, which typically requires two UPDATEs, is to debit account A and credit account B."
>
> "..., after you credit account B, you issue the COMMIT command, making the changes permanent. Only then do the changes become visible to other users."
>
> "..., if you cannot credit account B because of a logical problem, you execute the ROLLBACK statement. This undoes the change to account A, thereby restoring the original data."

Now, imagine a database table `BankAccounts` defined by:

```
/* Bank accounts are identified by unique Id's */
CREATE TABLE BankAccounts (
  Id <INTERNAL> NOT NULL,
  Balance MONEY NOT NULL
);
CREATE UNIQUE INDEX BankAccounts_IX1 ON BankAccounts(Id);
```

Then, here is how *I* would write the transaction (an ORACLE SQL*Plus script using PL/SQL):

```
DECLARE
  unable_to_deposit  EXCEPTION;
  unable_to_withdraw EXCEPTION;
BEGIN
  -- Transfer $100 from bank account <A> to bank account <B>
  UPDATE BankAccounts
   SET   Balance=Balance-100  -- Withdraw $100
   WHERE Id=<A>
   AND   Balance>=100;        -- , if possible
  IF SQL%ROWCOUNT<>1 THEN
    RAISE unable_to_withdraw;
  END IF;
  UPDATE BankAccounts
   SET   Balance=Balance+100  -- Deposit $100
   WHERE Id=<B>;
  IF SQL%ROWCOUNT<>1 THEN
    RAISE unable_to_deposit;
  END IF;
  COMMIT; -- End transaction making the changes permanent
EXCEPTION
  WHEN unable_to_withdraw OR unable_to_deposit THEN
    ROLLBACK; -- End transaction undoing any changes
END;
/
```

Note: In the above example `BEGIN` and `END` do *not* mark the beginning and end of the transaction. They mark the beginning and end of the PL/SQL-block. The transaction associated with the SQL*Plus-script above is implicitly begun before execution of the first SQL-statement (`UPDATE` ...) and is ended explicitly with the `COMMIT` or the `ROLLBACK`.

This is how the Bank Account example is presented in [52, p.145]:

> "Now look at a modern banking application that updates an on-line data base in place. The customer calls up the bank using a PC with a modem with the intention of withdrawing money from one account and depositing it in another. The operation is performed in two steps:
>
> 1. Withdraw(amount,account1).
> 2. Deposit(amount,account2).
>
> If the telephone line is broken after the first one but before the second one, the first account will have been debited but the second one will not have been credited. The money vanishes into thin air.
>
> Being able to group these two operations in an atomic transaction would solve the problem. Either both would be completed, or neither would be completed. The key is rolling back to the initial state if the transaction fails to complete."

and [52, p.225-226]:

> "The classical example of where transactions make programming much easier is in a banking system. Imagine that a certain bank account contains 100 dollars, and that two processes are each trying to add 50 dollars to it. In an unconstrained system, each process might simultaneously read the file containing the current balance (100), individually compute the new balance (150), and successively overwrite the file with this new value. The final result could either be 150 or 200, depending on the precise timing of the reading and writing. By grouping all the operations into a transaction, interleaving cannot occur and the final result will always be 200."

The two different situations could be sketched:

Situation 1 - Interleaving processes

Process 1	Process 2
READ(amount,account)	
	READ(amount,account)
WRITE((amount+$50),account)	
	WRITE((amount+$50),account)
...	
	...

Situation 2 - Isolated transactions

Process 1	Process 2
BEGIN_TRANSACTION READ(amount,account) WRITE((amount+$50),account) END_TRANSACTION	
	BEGIN_TRANSACTION READ(amount,account) WRITE((amount+$50),account) END_TRANSACTION

make

This example is taken from [30]:

> "Consider the following scenario of a partitioned read/write conflict. A programmer Joe caches relevant files on his Coda laptop for a weekend trip. While disconnected, he edits some source files and builds a new version of `repair`, a file resolution program. However one of the libraries `libresolve.a` that is linked in was updated on the servers during Joe's absence. Here the linking and the updating of `libresolve.a` constitute a partitioned read/write conflict, which not only leaves `repair` in a possible inconsistent state but also may cause cascading inconsistencies had Joe used his `repair` program to mutate other objects. It would be helpful if Joe is at least notified about the possible inconsistency when he reconnects the laptop to the servers."

Other references to this scenario can be found in [34] and in [20]. A similar scenario is used in [31].

Mail Reader

An existing application used directly or ported, e.g., a mail reader:

ftp://ftp.diku.dk/pub/linux/slackware/source/n/elm/elm2.4.tar.gz

Blackboard

This "Real World Example" comes from [2]:

> "A blackboard is used for leaving messages to people.
>
> When we visit the blackboard we can read from it and write to it (and make a photo of it).
>
> When we are left (disconnected) we can watch an old photo of the blackboard and write messages onto the photo.
>
> When we visit the blackboard we can decide which of the messages on the photo that are still actual to write to the blackboard.
>
> Too old photos are not acceptable and to long time until coming back to the blackboard is not acceptable.
>
> -> Time seems in most cases important!
>
> -> The world consists of cooperative processes!"

A "Class Example" is also given:

```
class Blackboard(rows,columns: Integer)->(self: Blackboard)
  const Interface:=[Write,Erase,Read]
  const ReadBound:=30*60 {seconds}
  const WriteBound:=20*60
  const Board:=Arrayd2D(rows,columns,var Char)

  function Read(x,y: Integer)->(t: Text)
    {read (local) board}
  end

  operation Erase()
    {erase (local) board}
  end

  operation Write(x,y: Integer, t: Text)
    {check for space and update (local) board}
  end

  operation MergeReplica(rep: Blackboard)
    {executed for primary replica}
  end
end
```

Note: Instead of making a photo when you visit (connect to) the blackboard (the server) you might read what is on the blackboard (shared data) and write a copy of what you find interesting (or might find interesting at a later stage) on your own miniature blackboard (client cache). When you are away (disconnected) from the blackboard, you can write messages (updates) onto the miniature blackboard.

While away, you might at some point think that the information on the miniature blackboard has become outdated and decide to phone (establish a weak connection) someone[1] who can confirm whether the information *is* out-of-date or not. If it is outdated, and you can afford the larger phone bill you can decide to retrieve the newest information and update the miniature blackboard. On the other hand if it was not long ago you left the blackboard or you do not have the strength to do an "update session" over the phone you might think that the information you have will suffice.

When you return (re-connect) to the blackboard, you decide to let one of your obedient students[2] transfer your writings on the miniature blackboard to the big blackboard. The student runs into problems if there is discrepancy between what is on the blackboard and what you have written. If the student is clever enough she solves any inconsistencies herself (automatically), but, if the problem is really bad, she might return to you and let you do it yourself (manually).

[1] In my case the server, so we have a sort of *talking* blackboard—NOW THAT IS SOMETHING NEW!

[2] The server again, this blackboard is ending up *very sophisticated*—INDEED!

Appendix C

Flow Diagrams

See attachment...

FLOW CHARTS

The following flow charts illustrate the decisions made during the execution of the PeStO file operations: `p_open`, `p_close`, `p_lock`, and `p_unlock`. The charts are not actual flow diagrams for the program (but close). The decisions are based on either simple questions that can be answered with “yes” or “no”, i.e.,

Is the file cached?

or the “success” or “failure” of communication with the server, i.e.,

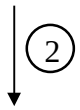
REQUEST STATUS FROM SERVER

Note, that an arrow pointing from a communication box, such as the one above, is not the server’s reply.

When an arrow points into a solid box, such as

OPEN FILE FOR WRITING (2)

then the flow branches, and in the above case the flow is continued on one of the pages labeled **OPEN FILE FOR WRITING** at the arrow marked with the number in the brackets, i.e.,



Questions labeled in italic and bold, i.e.,

Are we (fully) CONNECTED?

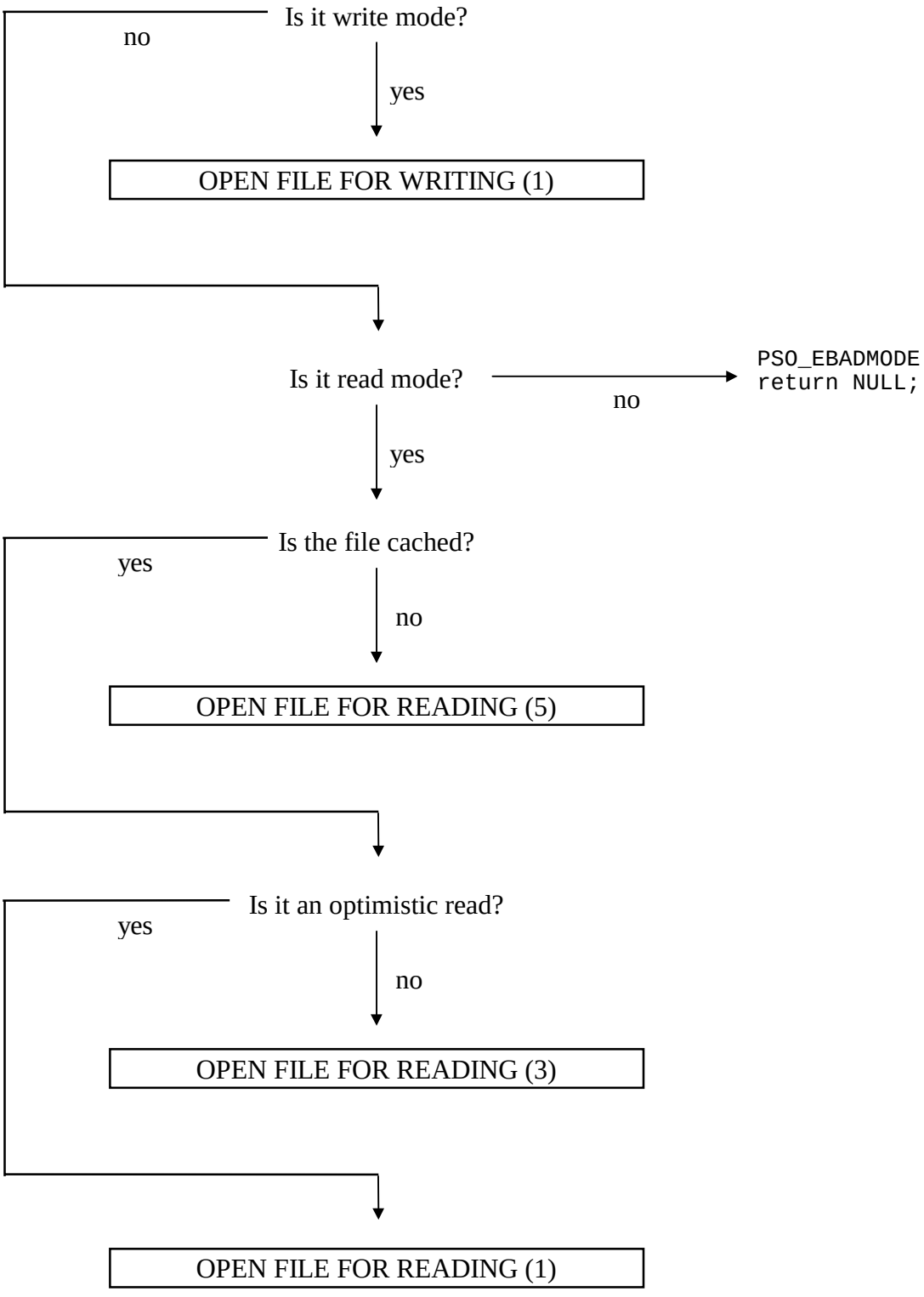
are answered by utilizing the TACO link monitoring facilities.

All flows end with a `return` command, that gives back control to the client application using the PeStO file operation. If an error occured, then the error number (`pso_errno`) is also given. Not all things are shown in the diagrams; technicalities such as what to be done when the file is not found, the fact that it is not necessary to receive the file from the server if it is going to be rewritten from scratch, and other details are **not** given.

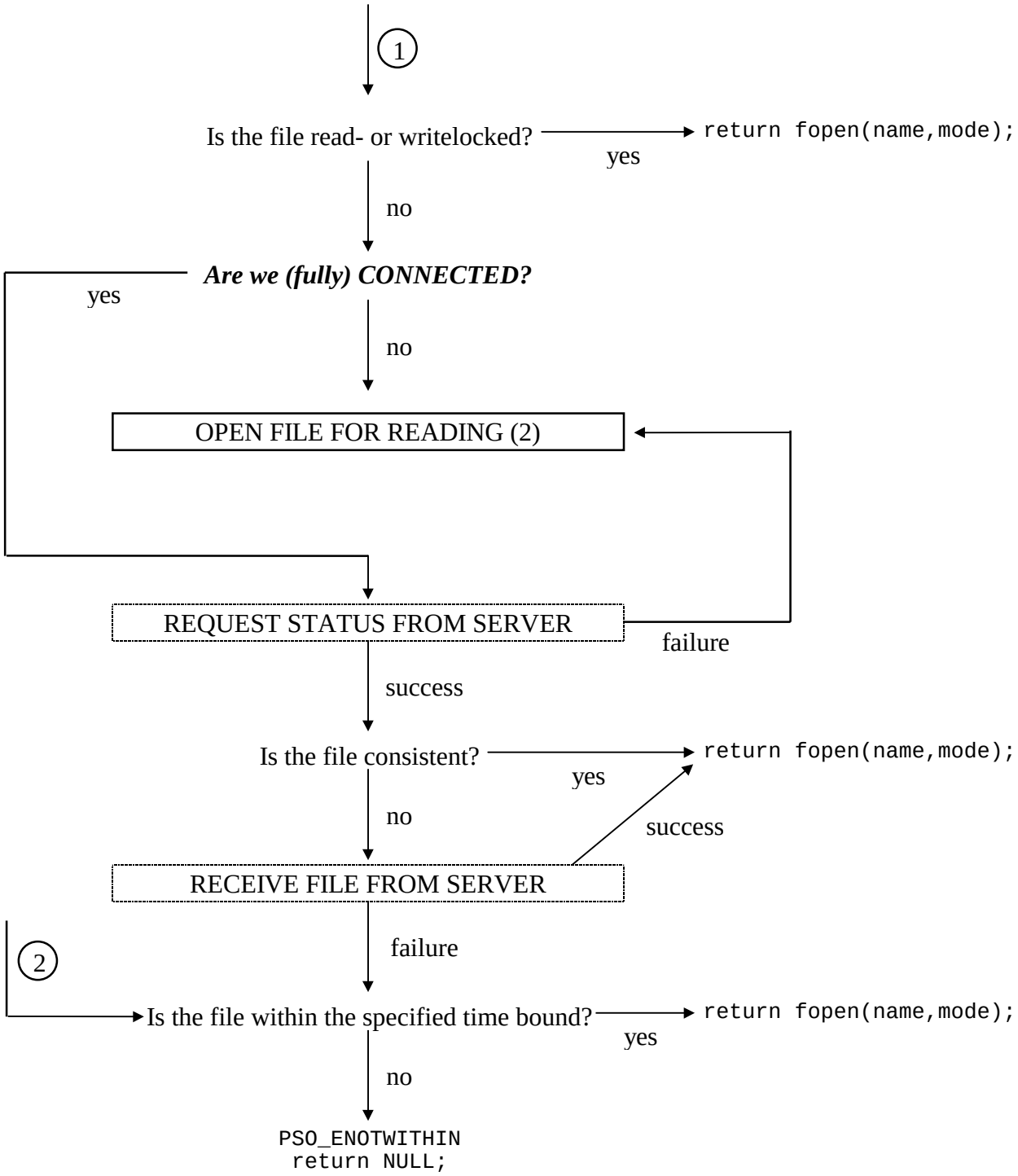
I think the charts are quite instructive and give a good insight to what actually goes on!

P_OPEN **START**

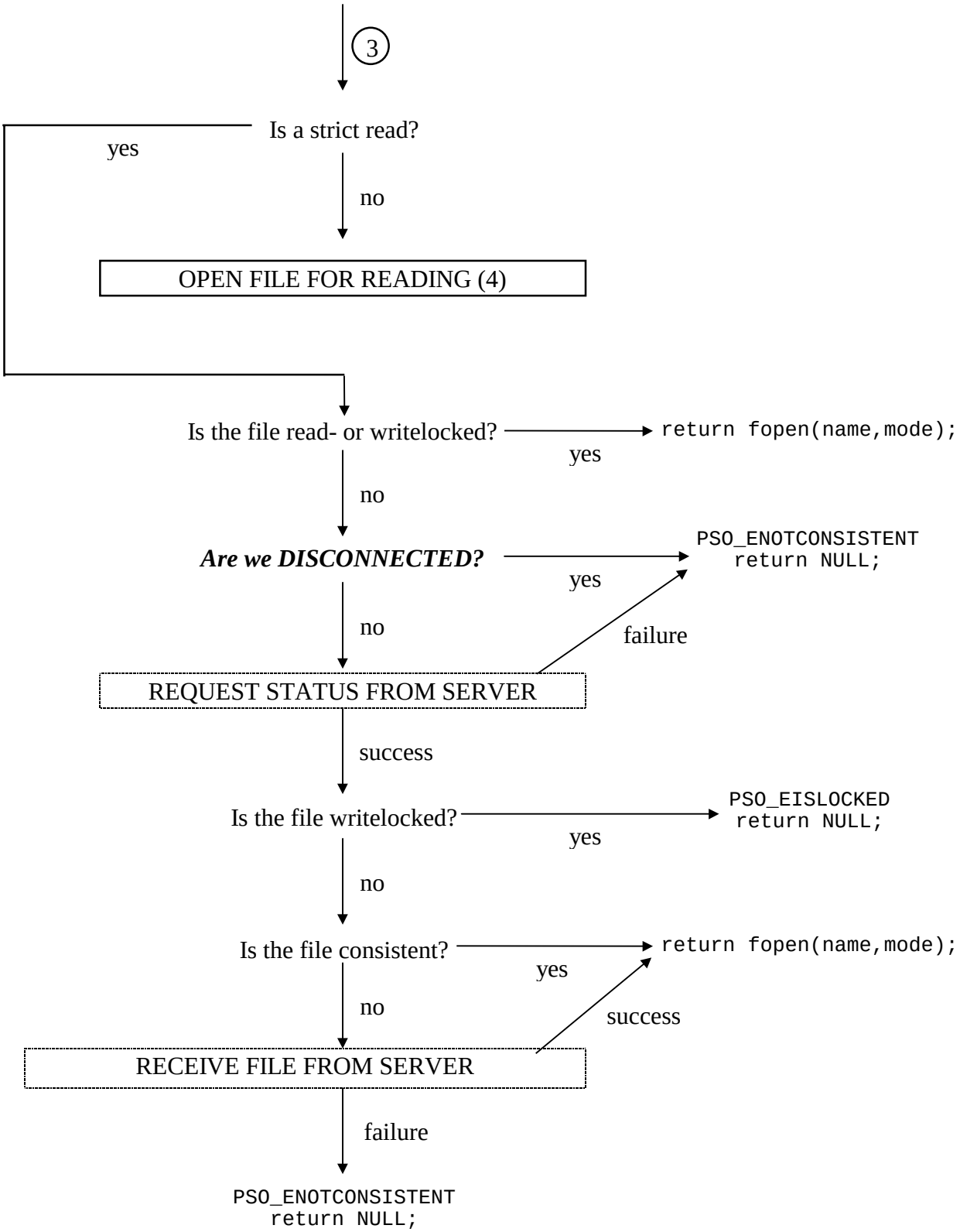
```
FILE *p_open(char *name,char *mode,int tb)
```



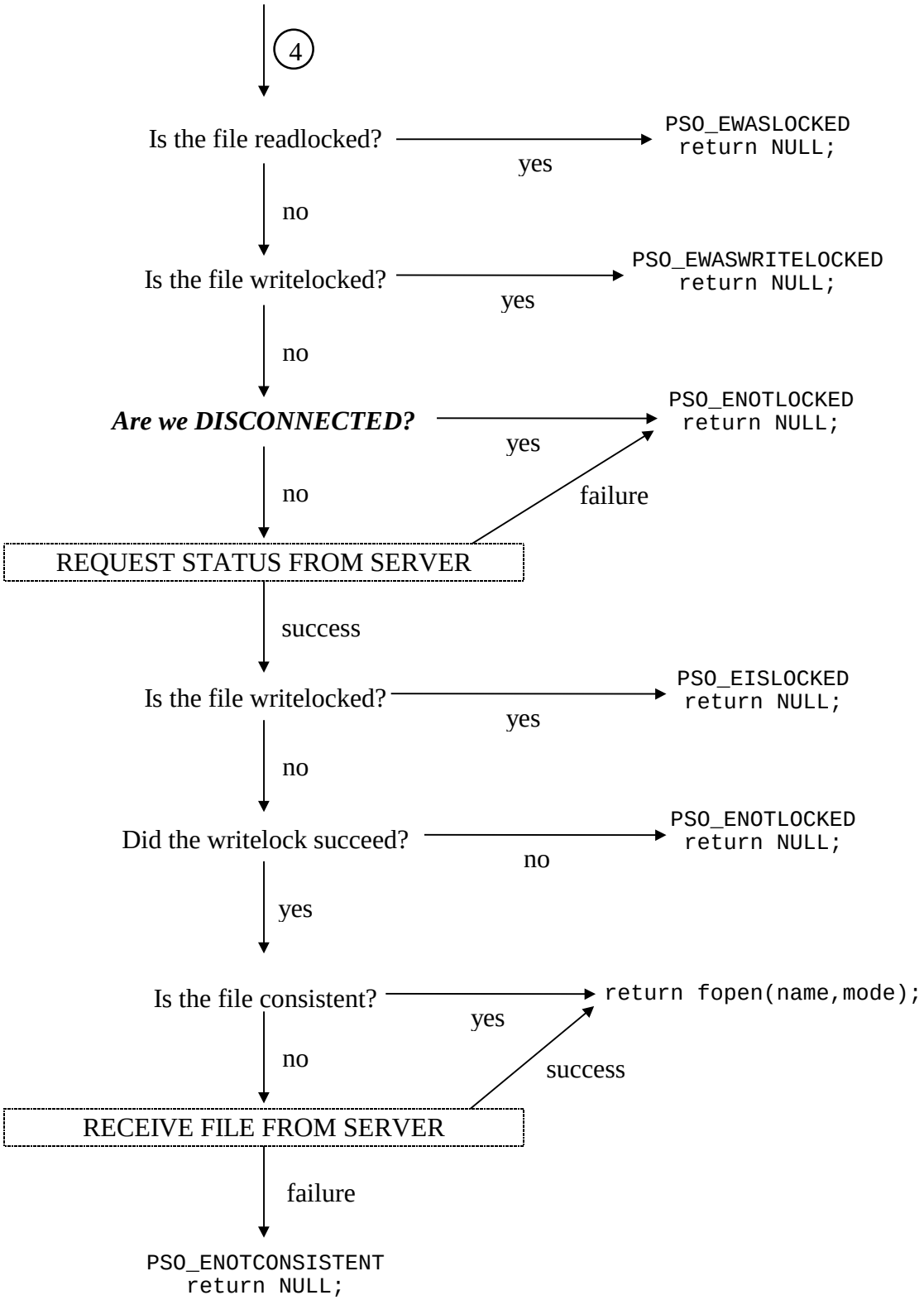
P_OPEN OPEN FILE FOR READING 1(4)



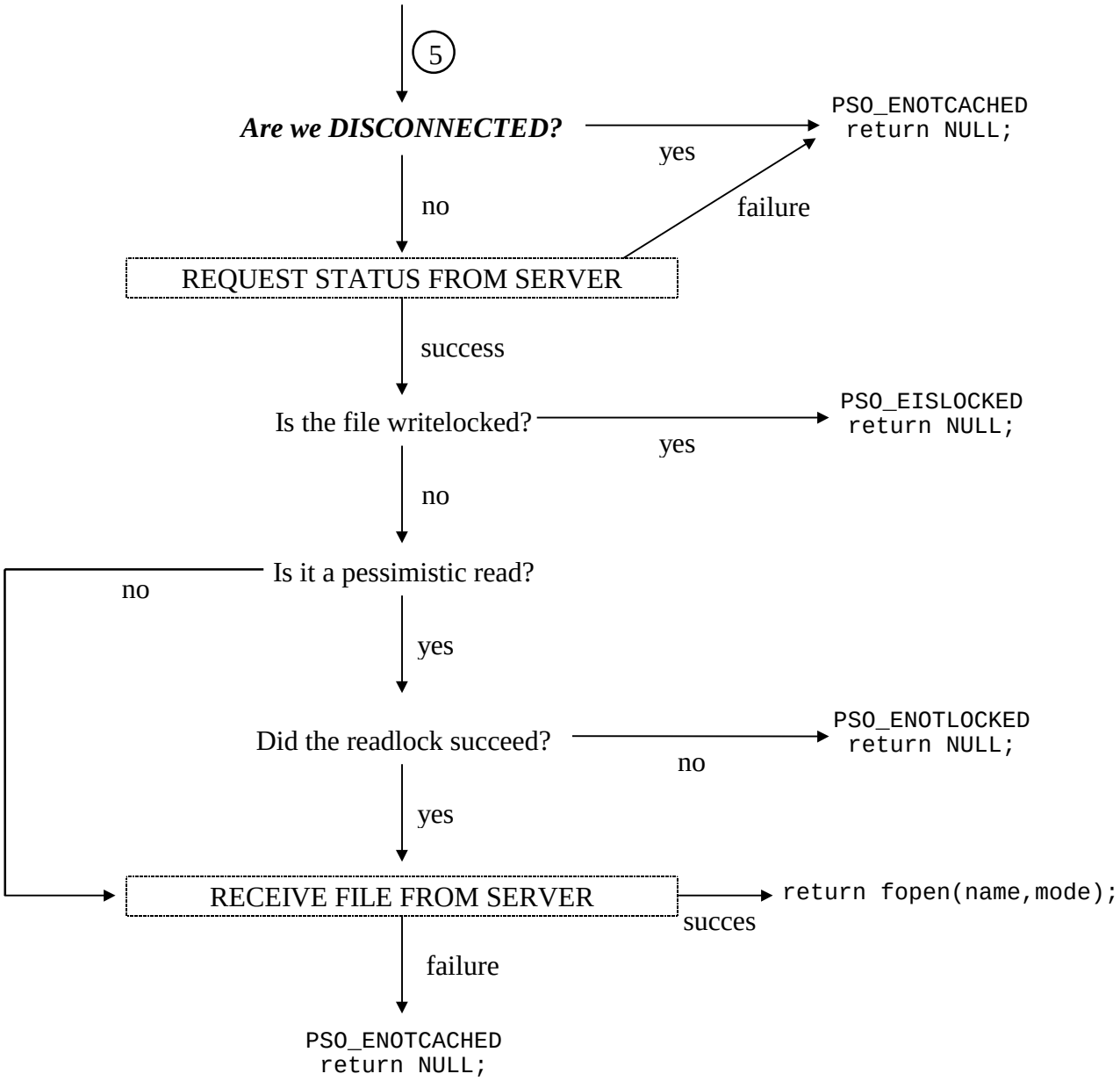
P_OPEN OPEN FILE FOR READING 2(4)



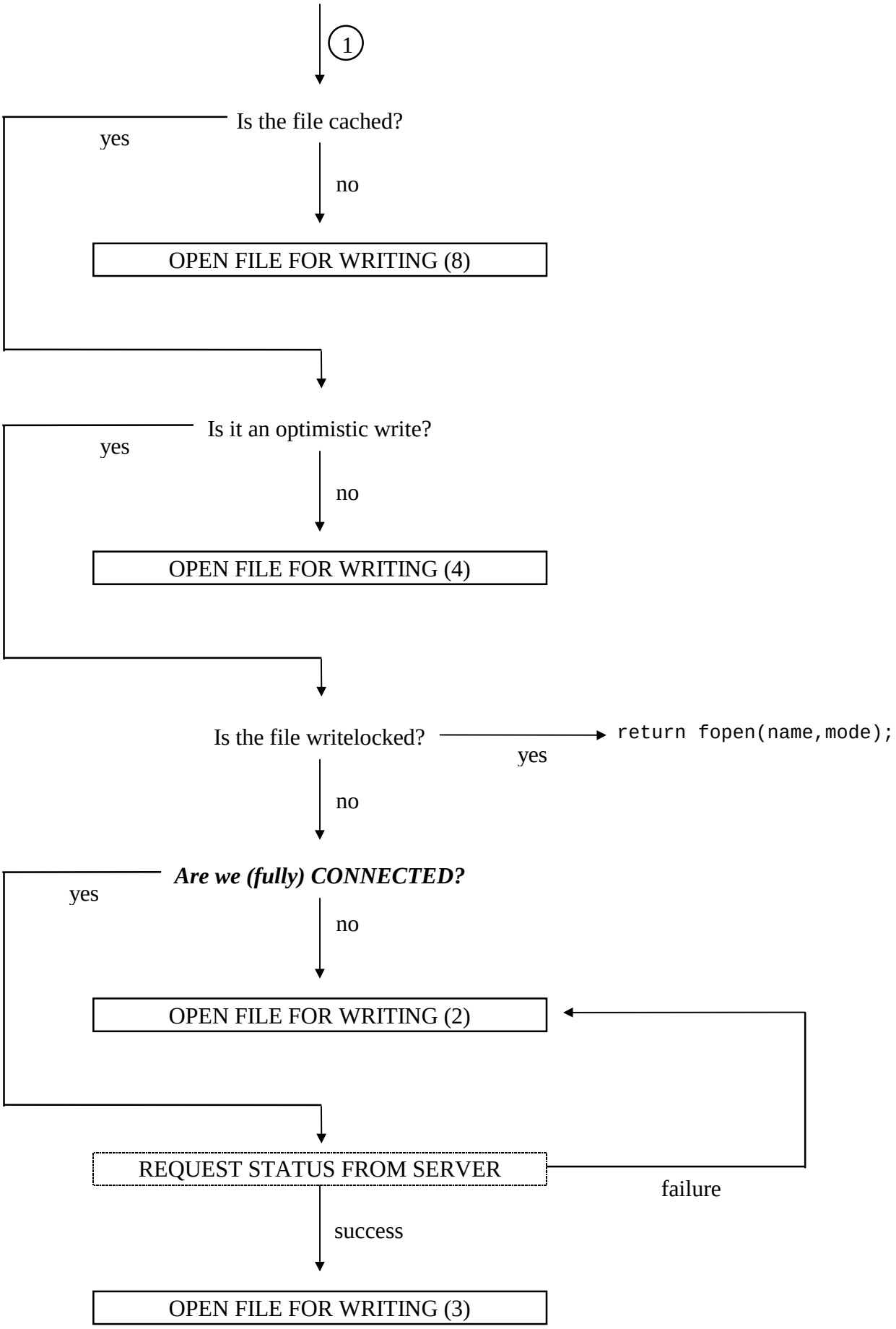
P_OPEN OPEN FILE FOR READING 3(4)



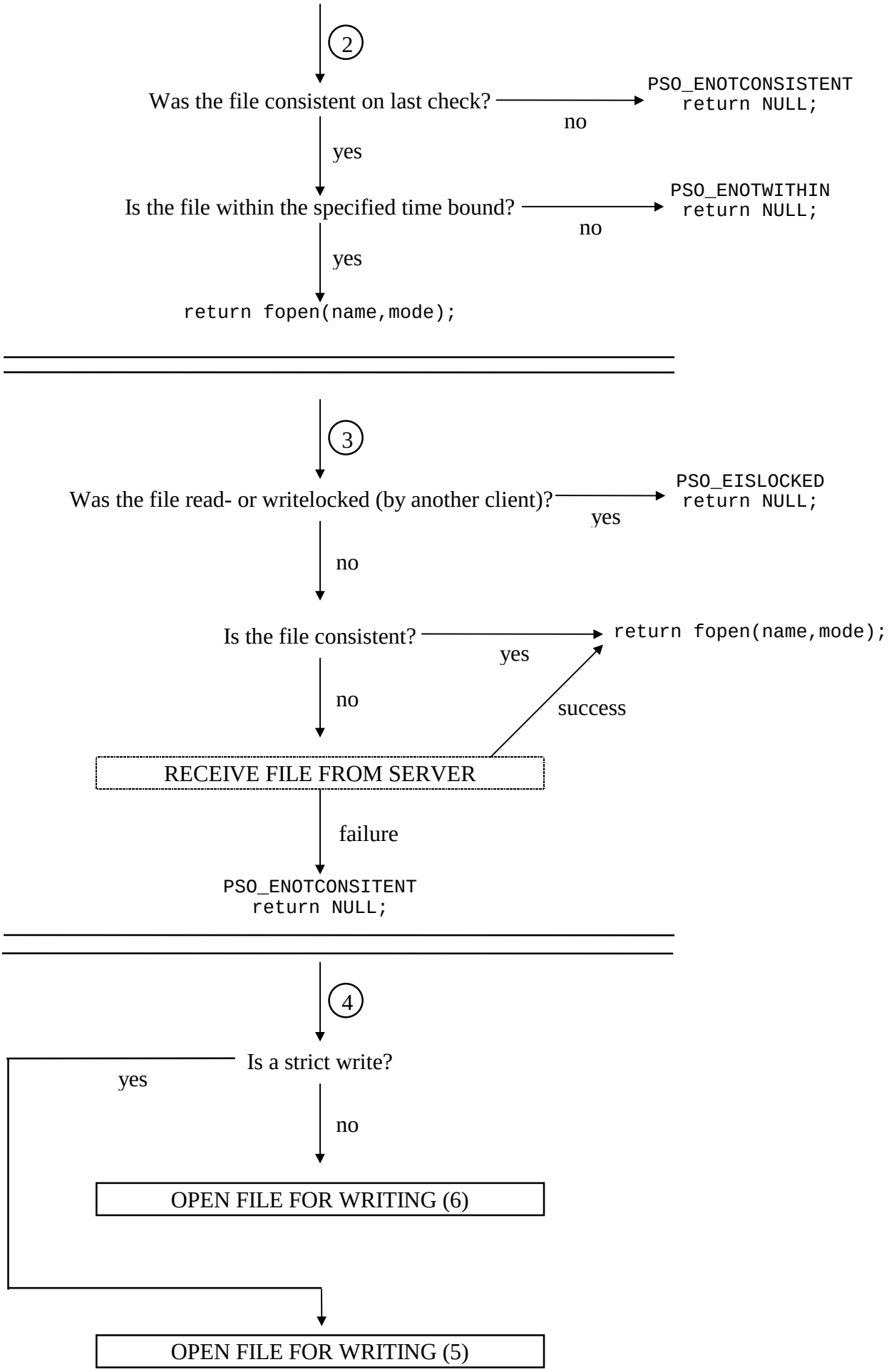
P_OPEN OPEN FILE FOR READING 4(4)



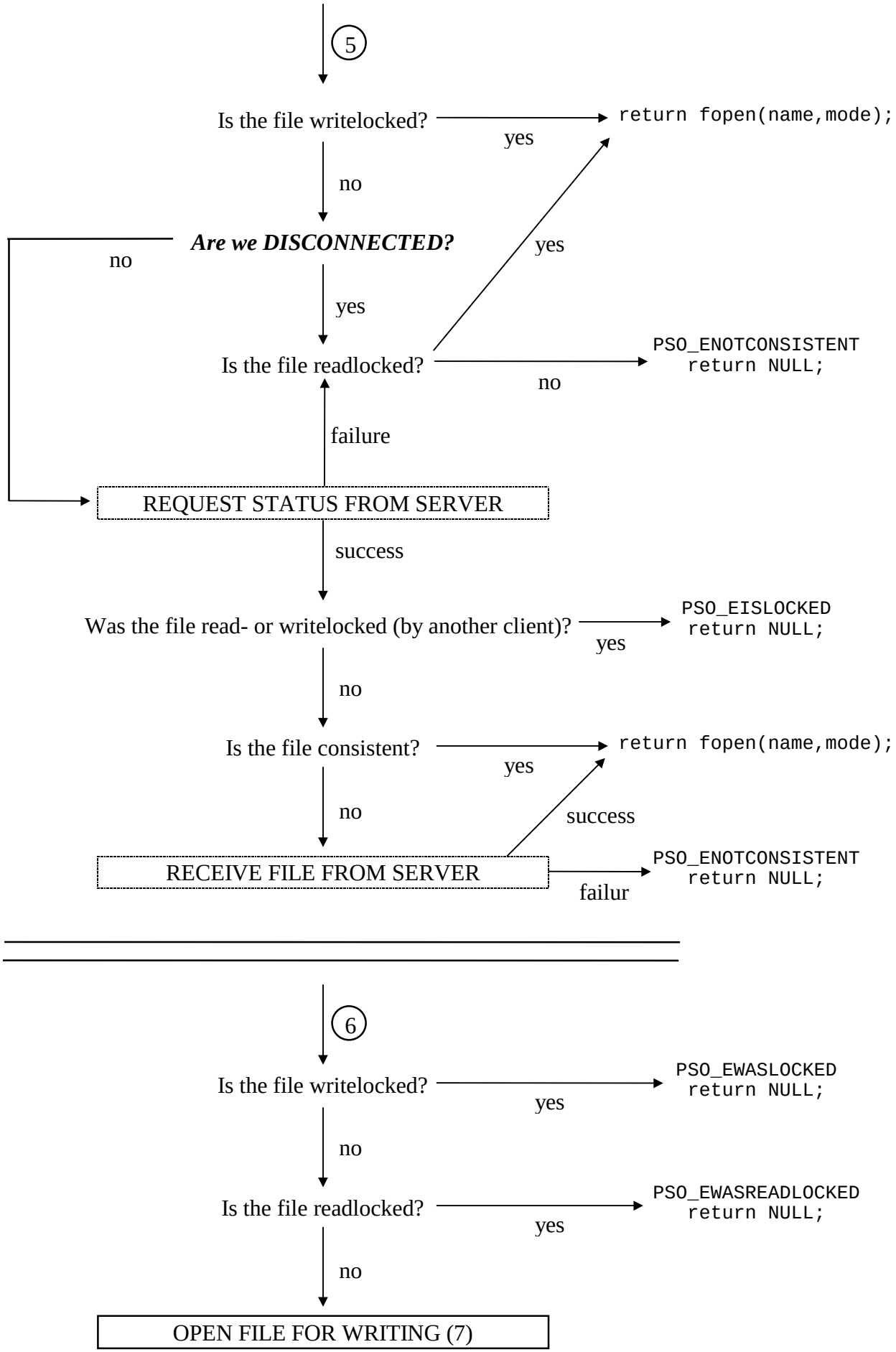
P_OPEN OPEN FILE FOR WRITING 1(5)



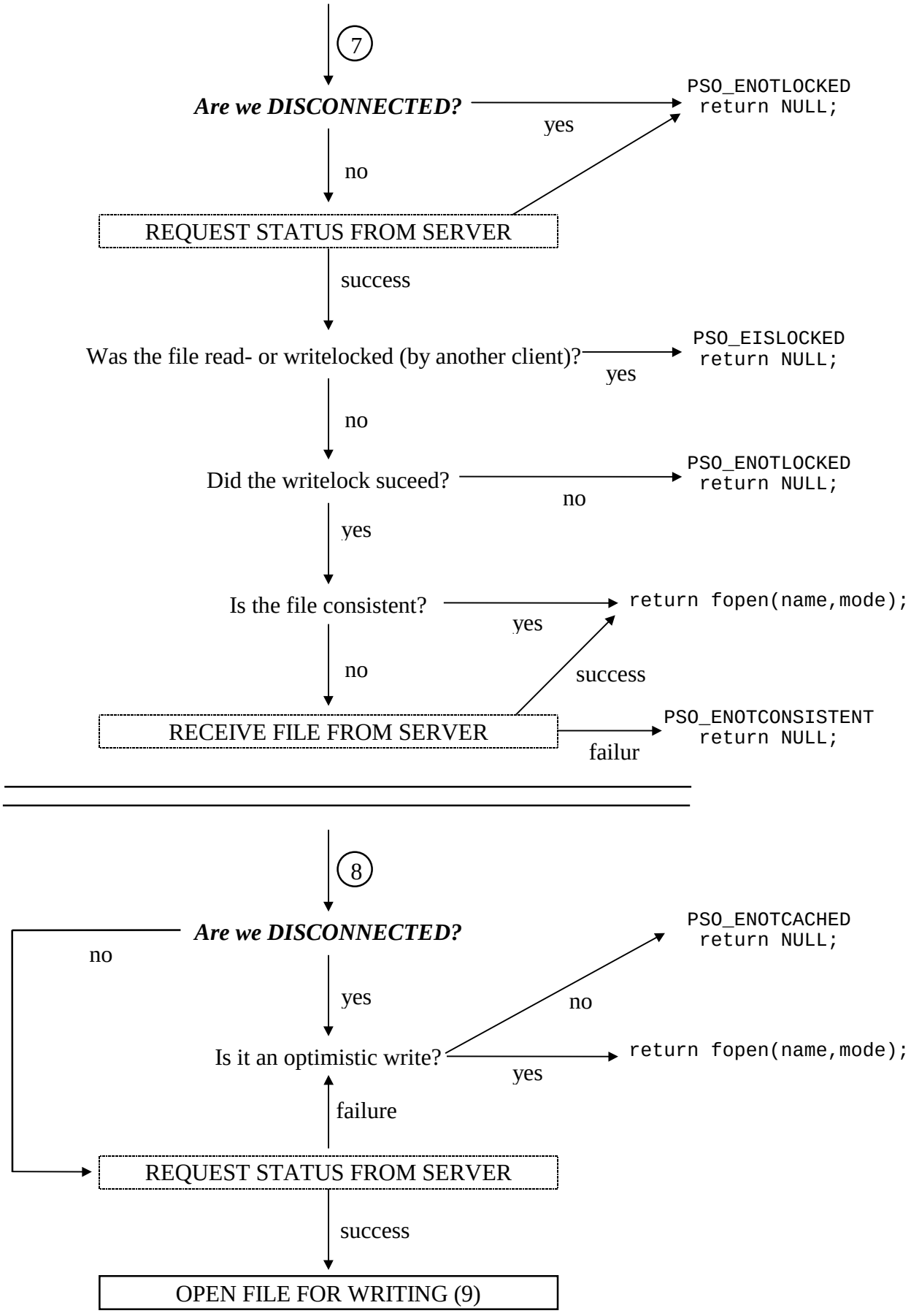
P_OPEN OPEN FILE FOR WRITING 2(5)



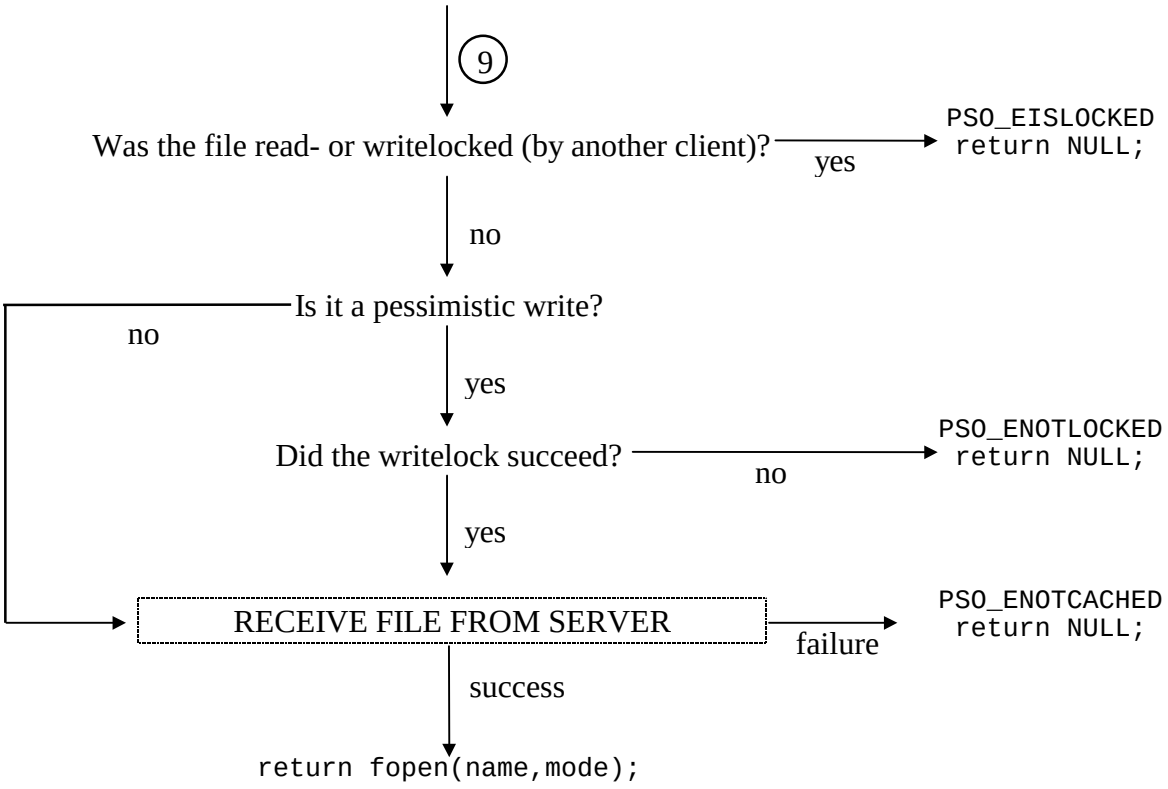
P_OPEN OPEN FILE FOR WRITING 3(5)



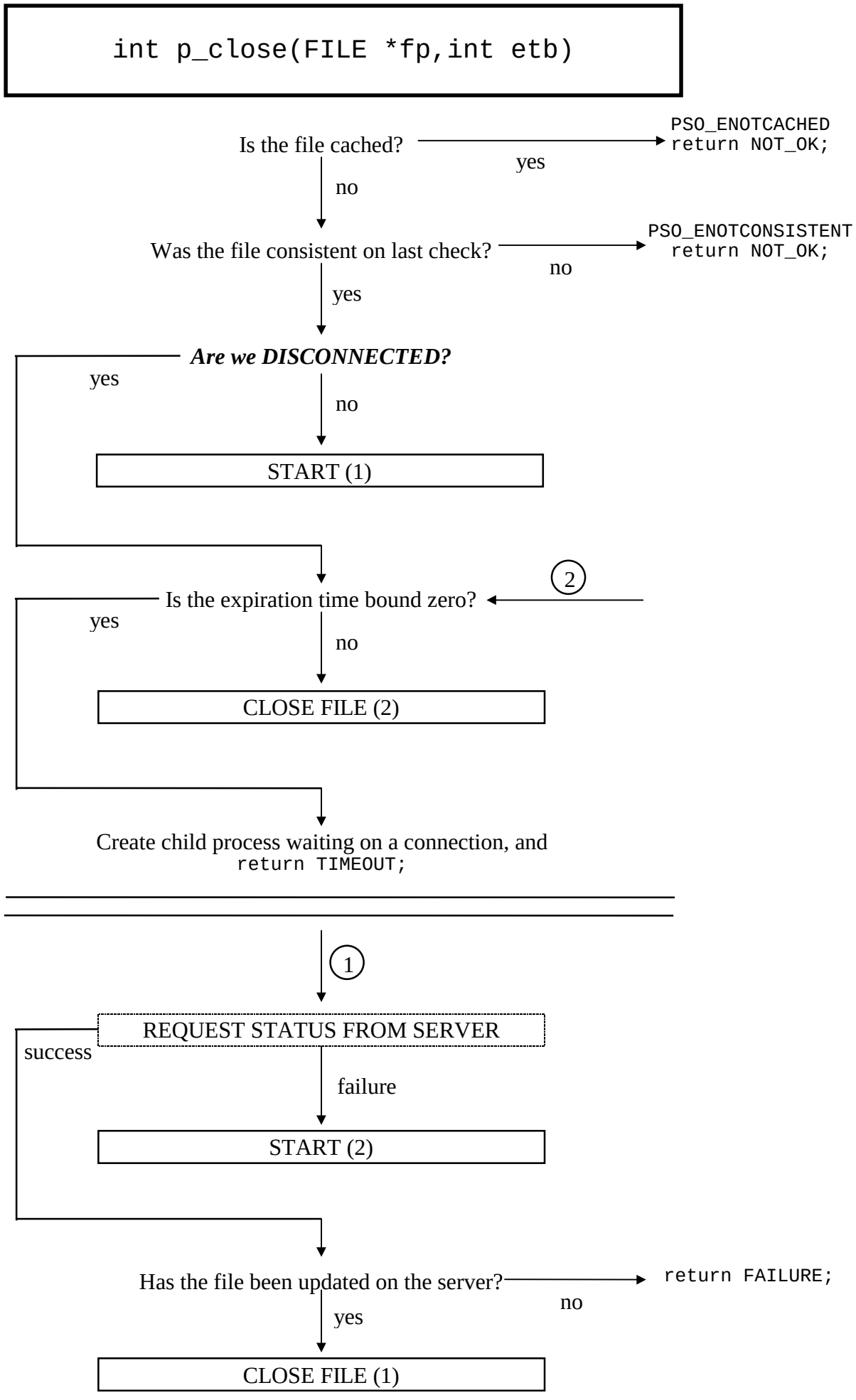
P_OPEN OPEN FILE FOR WRITING 4(5)



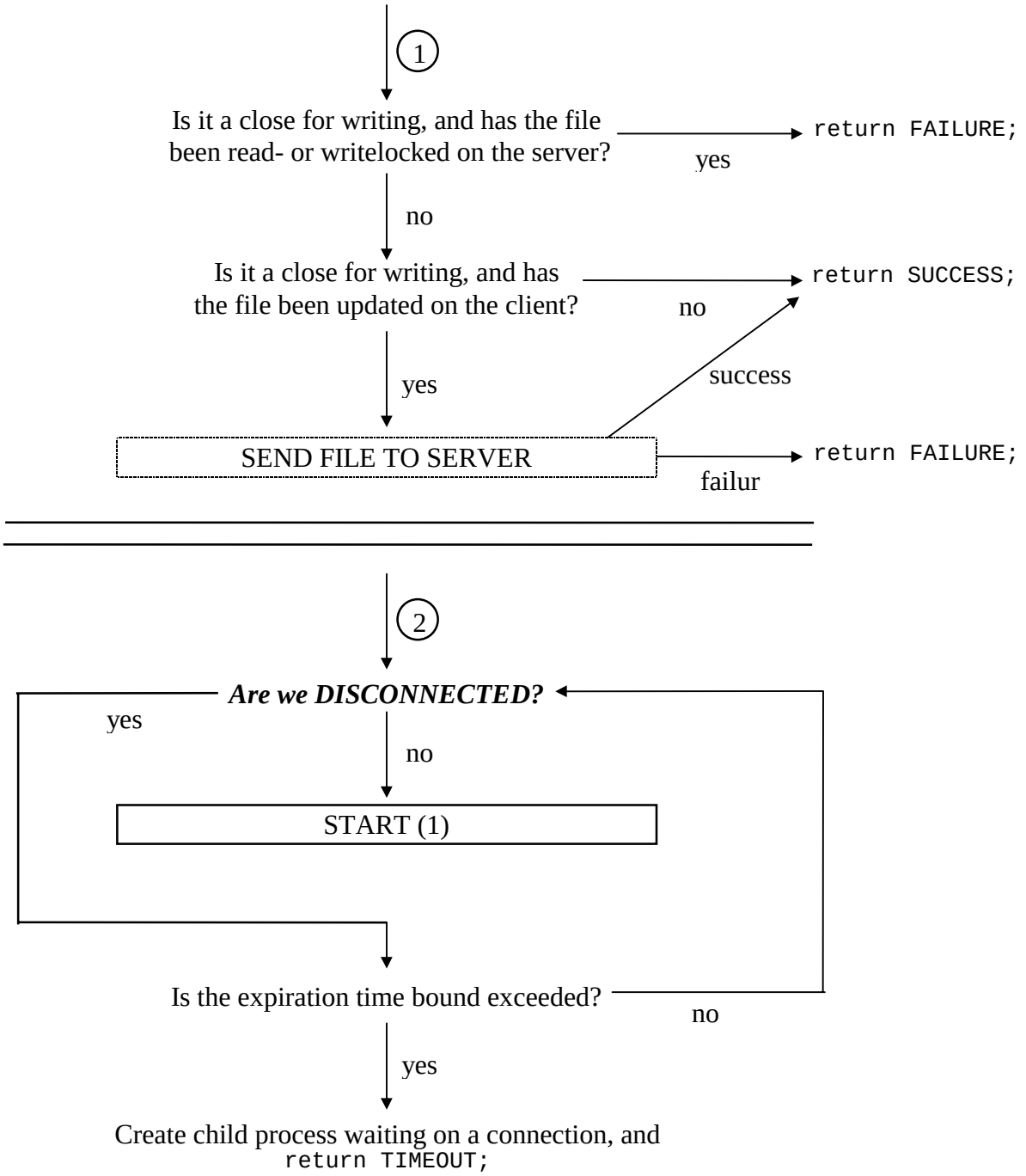
P_OPEN OPEN FILE FOR WRITING 5(5)



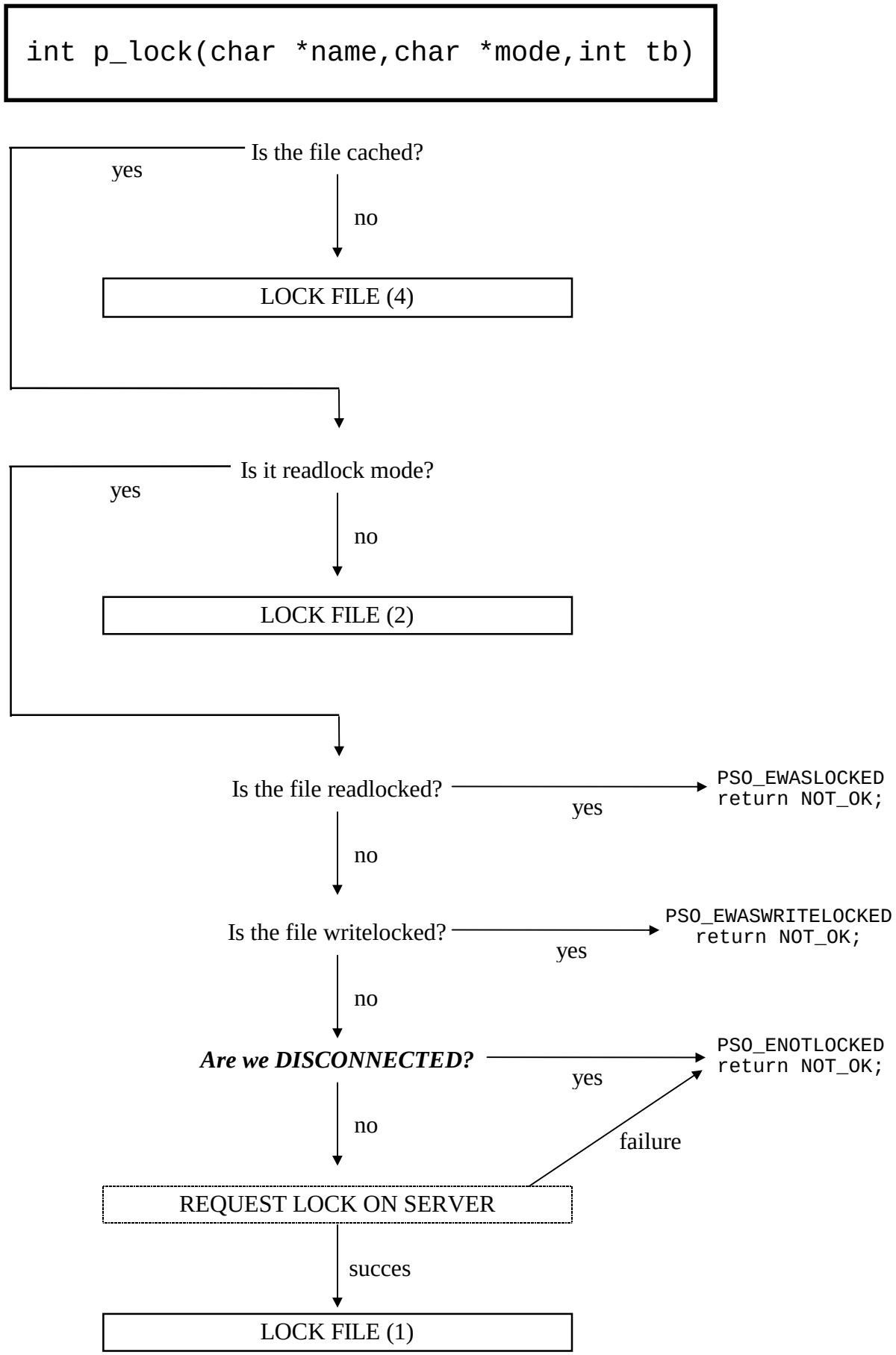
P_CLOSE **START**

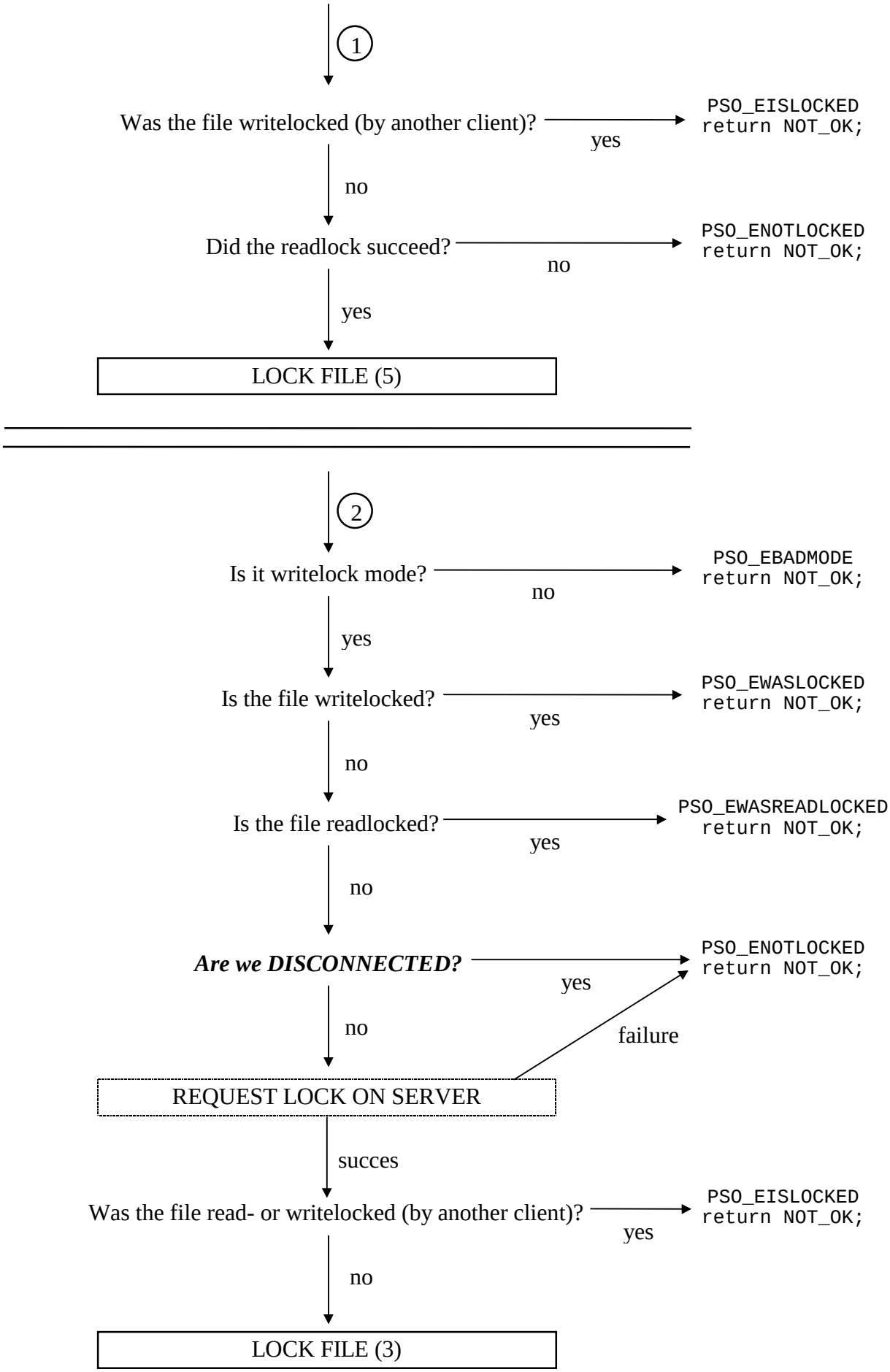


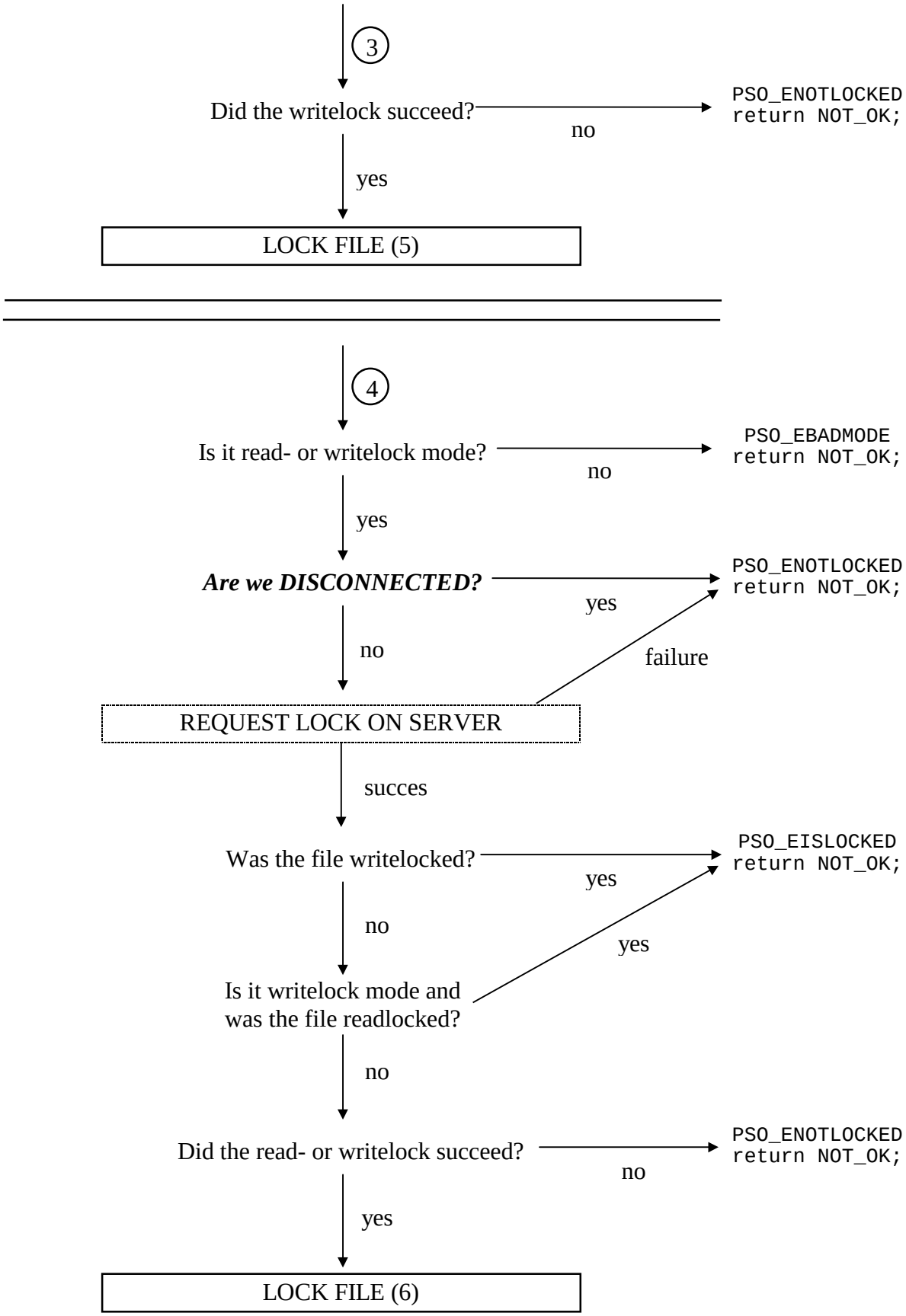
P_CLOSE **CLOSE FILE 1(1)**

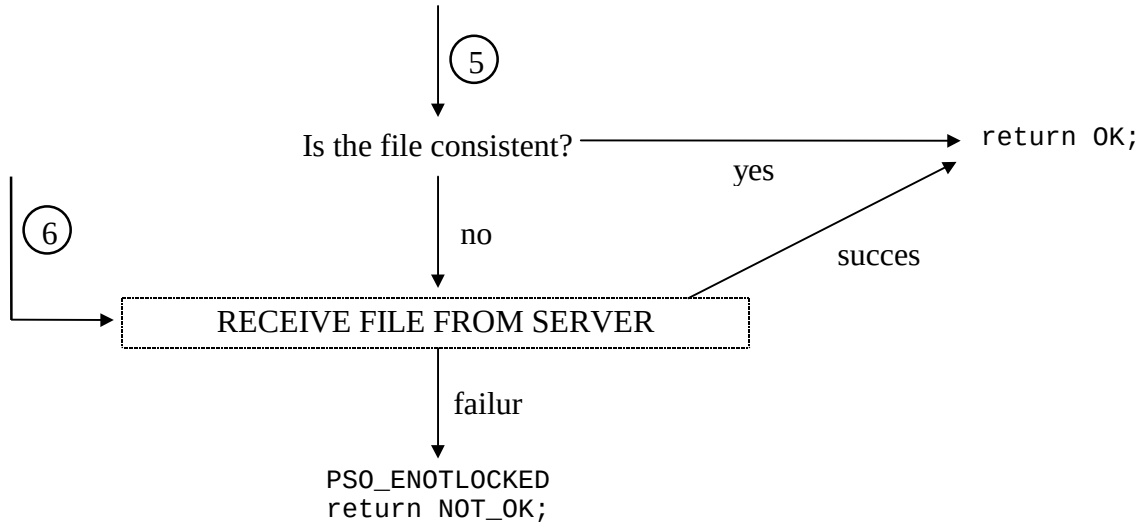


P_LOCK **START**

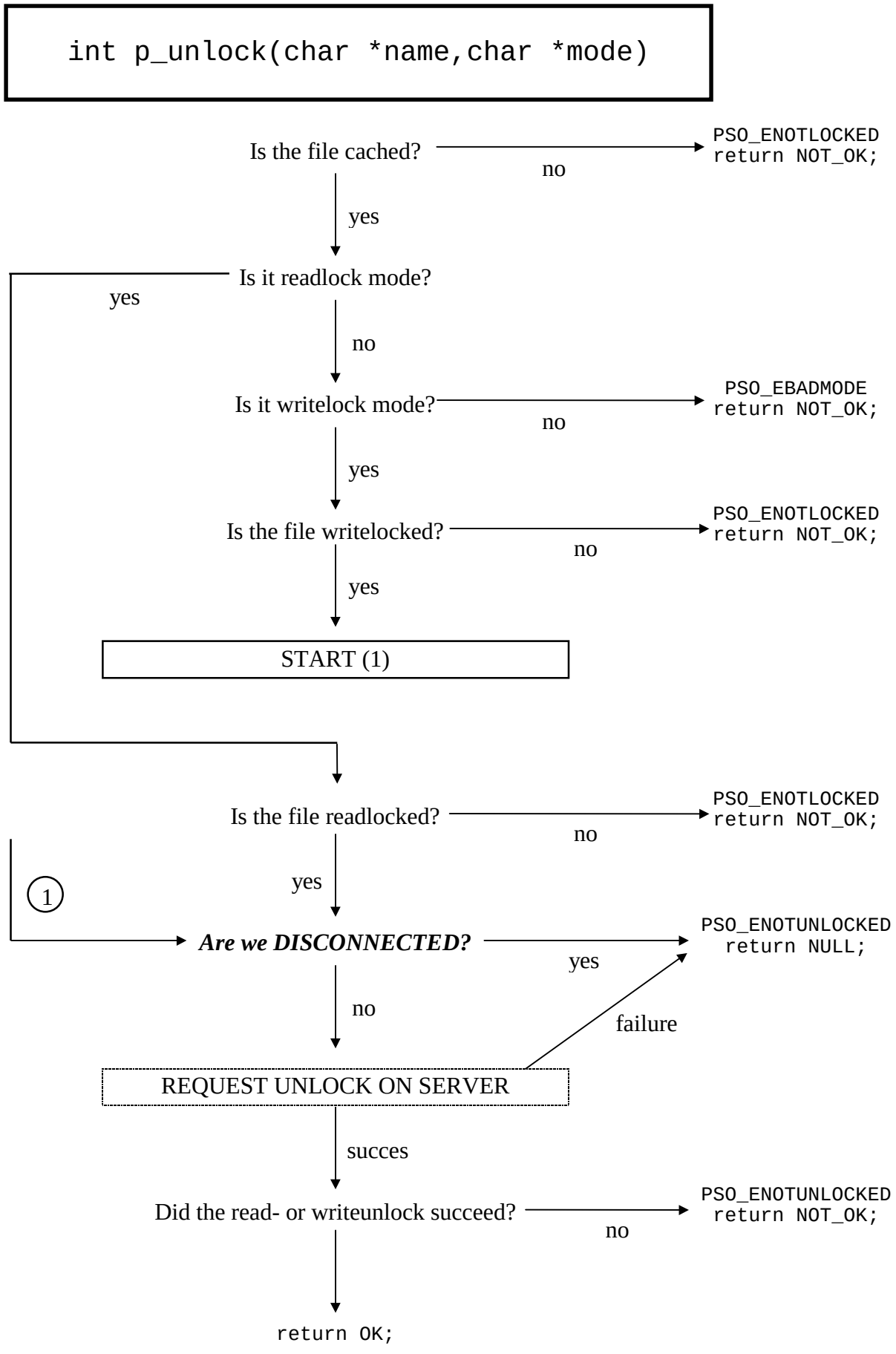

int p_lock(char *name,char *mode,int tb)
Is the file cached?
yes
no
LOCK FILE (4)
Is it readlock mode?
yes
no
LOCK FILE (2)
Is the file readlocked?
yes
PSO_EWASLOCKED
return NOT_OK;
no
Is the file writelocked?
yes
PSO_EWASWRITELOCKED
return NOT_OK;
no
Are we DISCONNECTED?
yes
PSO_ENOTLOCKED
return NOT_OK;
no
failure
REQUEST LOCK ON SERVER
succes
LOCK FILE (1)


1
Was the file writelocked (by another client)?
yes
PSO_EISLOCKED
return NOT_OK;
no
Did the readlock succeed?
no
PSO_ENOTLOCKED
return NOT_OK;
yes
LOCK FILE (5)
2
Is it writelock mode?
no
PSO_EBADMODE
return NOT_OK;
yes
Is the file writelocked?
yes
PSO_EWASLOCKED
return NOT_OK;
no
Is the file readlocked?
yes
PSO_EWASREADLOCKED
return NOT_OK;
no
Are we DISCONNECTED?
yes
PSO_ENOTLOCKED
return NOT_OK;
no
failure
REQUEST LOCK ON SERVER
succes
Was the file read- or writelocked (by another client)?
yes
PSO_EISLOCKED
return NOT_OK;
no
LOCK FILE (3)


3
Did the writelock succeed?
no
PSO_ENOTLOCKED
return NOT_OK;
yes
LOCK FILE (5)
4
Is it read- or writelock mode?
no
PSO_EBADMODE
return NOT_OK;
yes
Are we DISCONNECTED?
yes
PSO_ENOTLOCKED
return NOT_OK;
no
failure
REQUEST LOCK ON SERVER
succes
Was the file writelocked?
yes
PSO_EISLOCKED
return NOT_OK;
no
yes
Is it writelock mode and
was the file readlocked?
no
Did the read- or writelock succeed?
no
PSO_ENOTLOCKED
return NOT_OK;
yes
LOCK FILE (6)


5
Is the file consistent?
return OK;
yes
6
no
succes
RECEIVE FILE FROM SERVER
failur
PSO_ENOTLOCKED
return NOT_OK;

P_UNLOCK **START**



Appendix D

Figures from Chapter 5

See attachment...

Figure 5.4: No read/write conflict

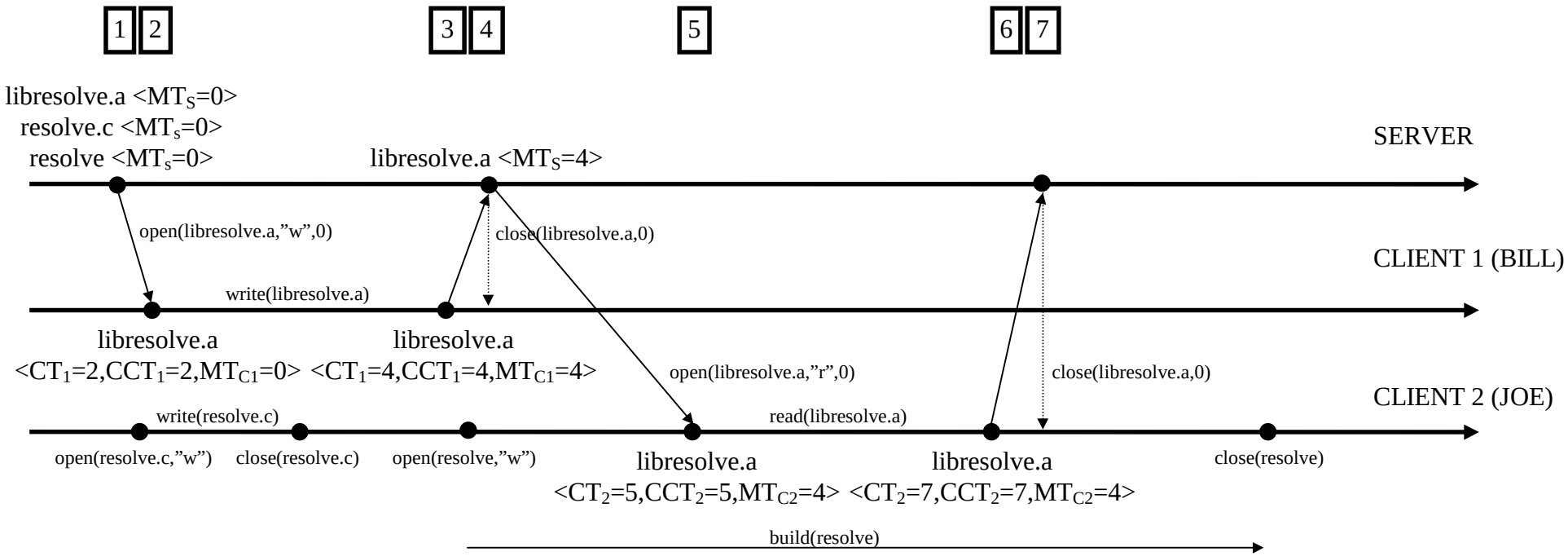


Figure 5.5: Undetactable read/write conflict

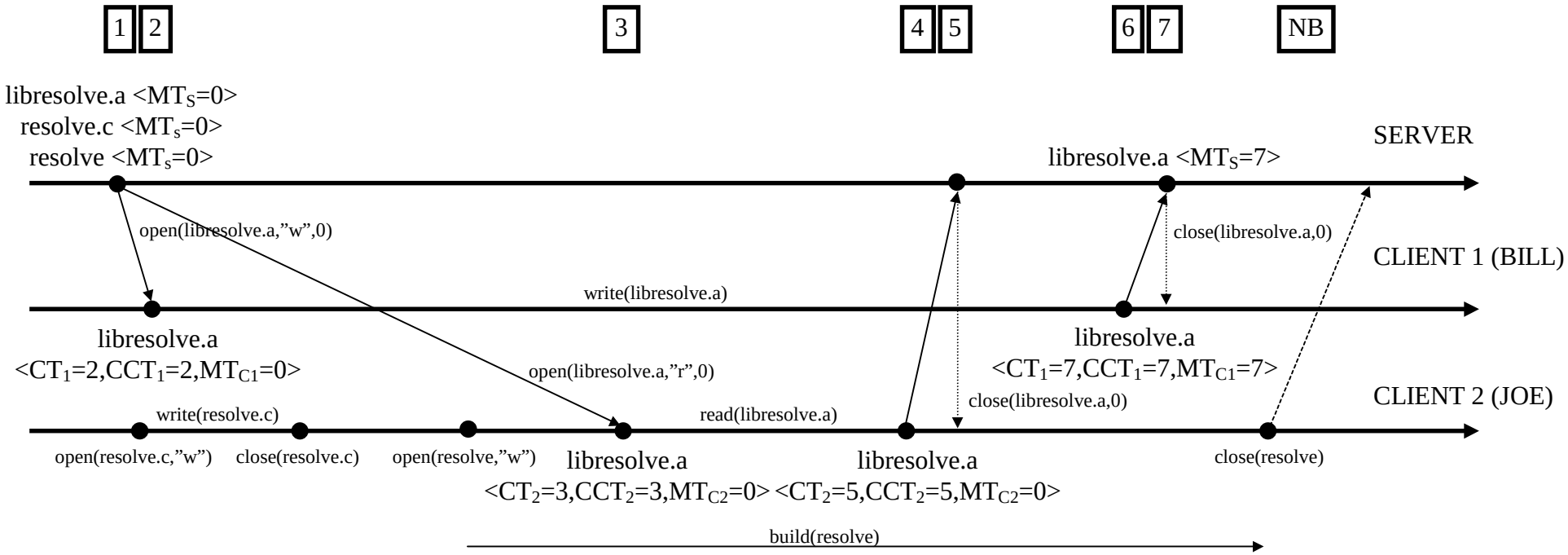
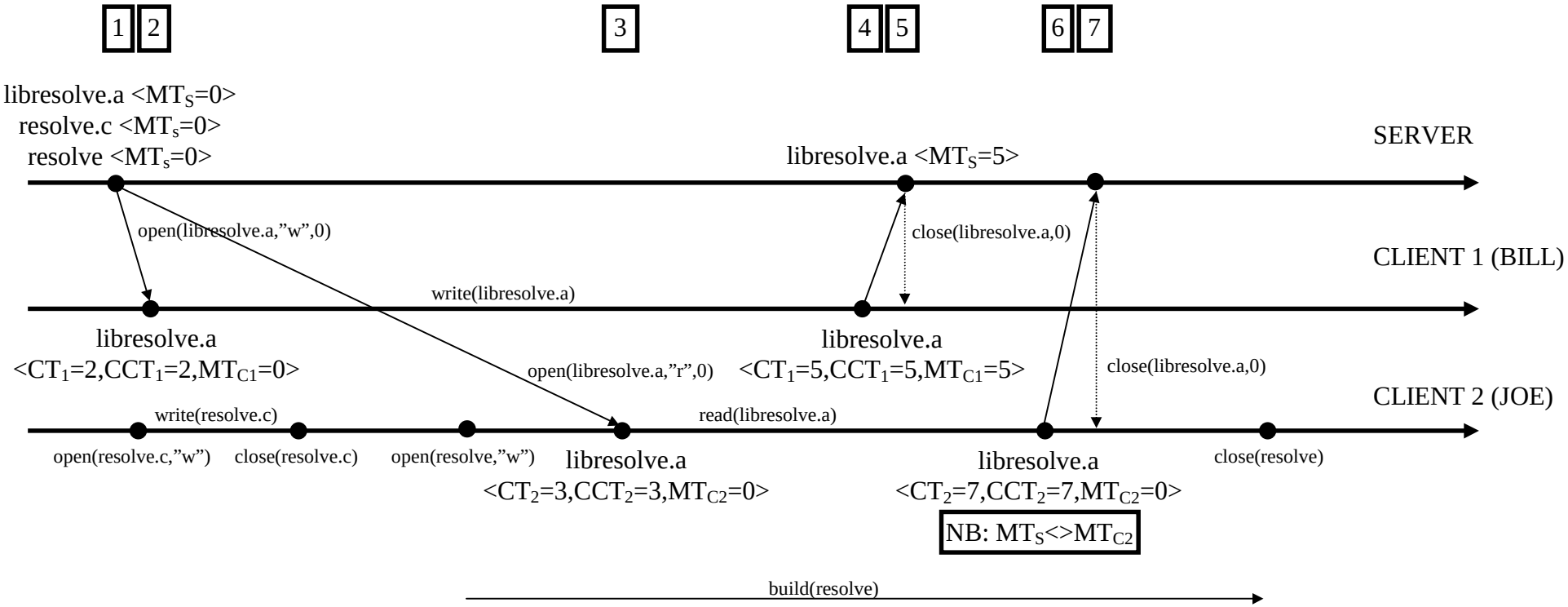


Figure 5.6: Detectable read/write conflict



References

[1] The **AMIGOS** Project:
<http://www.diku.dk/distlab/amigos/>
—as it appeared: December 1995

[2] Birger **Andersen**:
The Next Distributed Systems
Department of Computer Science, University of Copenhagen, Denmark
Notes: Braga, October 95

[3] Birger **Andersen**:
Personal e-mail correspondence
Department of Computer Science, University of Copenhagen, Denmark,
November 1996

[4] George **Coulouris**, Jean Dollimore & Tim Kindberg:
Distributed Systems - Concepts and Design
2nd edition, Addison-Wesley, 1994

[5] C. J. **Date**:
An Introduction to Database Systems
5th edition, Addison-Wesley, 1990

[6] Susan B. **Davidson**, Hector Garcia-Molina & Dale Skeen:
Consistency in Partitioned Networks
Computing Surveys, 17(3), September 1985

[7] Marc E. **Fiuczynski** & David Grove:
A Programming Methodology for Disconnected Operation
Department of Computer Science, University of Washington
March 8, 1994

[8] George H. **Forman** & John Zahorjan:
The Challenges of Mobile Computing

Department of Computer Science, University of Washington
UW CSE Tech Report #93-11-03, March 9, 1994

[9] Victor P. **Guedes** & Francisco Moura:
Replica Control in MIo-NFS
Deparmento de Informática,Universidade de Minho, Braga - Portugal
ECOOP'95 Workshop on Mobility and Replication
<ftp://ftp.diku.dk/diku/distlab/amigos/mionfs.ps>

[10] Jørgen Sværke **Hansen**, Torben Reich & Birger Andersen:
Semi-Connected TCP/IP in a Mobile Computing Environment
AMIGOS Position Paper
Department of Computer Science, University of Copenhagen, Denmark
<ftp://ftp.diku.dk/diku/distlab/amigos/sctcp.ps>

[11] Jørgen Sværke **Hansen** & Torben Reich:
Semi-Connected TCP/IP in a Mobile Computing Environment
Master's Thesis in Computer Science
Department of Computer Science, University of Copenhagen, Denmark
DIKU Project no. 95-6-11, June 10, 1996
<ftp://ftp.diku.dk/diku/distlab/amigos/diku95-6-11.ps.gz>

[12] Jørgen Sværke **Hansen**:
Users Guide for TACO
Department of Computer Science, University of Copenhagen, Denmark
DistLab Paper, November 15, 1996

[13] John S. **Heidemann** *et al.*:
Primarily Disconnected Operation: Experiences with Ficus
Department of Computer Science, University of California, Los Angeles
Proc. of the 2nd Workshop on Management of Replicated Data, IEEE, November 1992

[14] Peter **Honeyman**:
Taking a Little Work *Along*
Center for Information Technology Integration, University of Michigan, Ann Arbor
CITI Technical Report 91-5, August, 1991
<ftp://citi.umich.edu/pub/techreports/citi-tr-91-5.ps.Z>

[15] Peter **Honeymann**, Larry Huston, Jim Rees & Dave Bachmann:
The Little Work *Project*

Proc. of the 3rd IEEE Workshop on Workstation Operating Systems, April 1992

[16] L. B. **Huston** & P. Honeymann:
Disconnected Operation for AFS
Center for Information Technology Integration, University of Michigan, Ann Arbor
CITI Technical Report 93-3, June 18, 1993
<ftp://citi.umich.edu/pub/techreports/citi-tr-93-3.ps.Z>

[17] L. B. **Huston** & P. Honeymann:
Peephole Log Optimization
Center for Information Technology Integration, University of Michigan, Ann Arbor
CITI Technical Report 95-3, January 26, 1995
<ftp://citi.umich.edu/pub/techreports/citi-tr-95-3.ps.Z>

[18] L. B. **Huston** & P. Honeymann:
Partially Connected Operation
Center for Information Technology Integration, University of Michigan, Ann Arbor
CITI Technical Report 95-5, May 25, 1995
<ftp://citi.umich.edu/pub/techreports/citi-tr-95-5.ps.Z>

[19] L. B. **Huston** & P. Honeymann:
Communication and Consistency in Mobile File Systems
Center for Information Technology Integration, University of Michigan, Ann Arbor
CITI Technical Report 95-11, October 5, 1995
<ftp://citi.umich.edu/pub/techreports/citi-tr-95-11.ps.Z>

[20] Cristian **Ionitoiu**:
Mobile Agents Based Mobile Computing
Computer Science Department, Politechnica University of Timisoara
Slides: University of Copenhagen, May 1996

[21] Brian W. **Kernighan** & Dennis M. Ritchie:
The C (ANSI-C) Programming Language
2nd edition, Prentice Hall, 1988.

[22] James J. **Kistler** & M. Satyanarayanan:
Disconnected Operation in the Coda File System

Carnegie Mellon University
ACM Transactions on Computer Systems, 10(1):3-25, February 1992

[23] Pernille **Knudsen** & Michael G. Sørensen:
Distribueret filsystem
Datalogisk Institut, Københavns Universitet, November 1994

[24] Henning **Koch**, Lars Krombholz & Oliver Theel:
A Brief Introduction into the World of 'Mobile Computing'
Department of Computer Science, University of Darmstadt, Germany
THD-BS-1993-03, May 21, 1993

[25] Geoffrey H. **Kuenning**, Gerald J. Popek, Peter L. Reiher:
An Analysis of Trace Data for Predictive File Caching in Mobile Computing
Computer Science Department, University of California, Los Angeles
Technical Report CSD-940016, April 1994

[26] Geoffrey H. **Kuenning**:
The Design of the Seer Predictive Caching System
Computer Science Department, University of California, Los Angeles
IEEE Workshop on Mobile Computing Systems and Applications, 1994

[27] Puneet **Kumar** & M. Satyanarayanan:
Log-Based Directory Resolution in the Coda File System
School of Computer Science, Carnegie Mellon University
CMU-CS-91-164, December 14, 1991
<ftp://reports.adm.cs.cmu.edu/usr/anon/1991/CMU-CS-91-164.ps>

[28] Puneet **Kumar** & M. Satyanarayanan:
Flexible and Safe Resolution of File Conflicts
School of Computer Science, Carnegie Mellon University
CMU-CS-94-214, November 1994
<ftp://reports.adm.cs.cmu.edu/usr/anon/1994/CMU-CS-94-214.ps>

[29] H. T. **Kung** & John T. Robinson:
On Optimistic Methods for Concurrency Control
ACM Transactions on Database Systems, 6(2):213-226, June 1981

[30] Qi **Lu** & M. Satyanarayanan:
Isolation-Only Transactions for Mobile Computing
School of Computer Science, Carnegie Mellon University
ACM Operating Systems Review, 28(2):81-87, April 1994

[31] Qi **Lu** & M. Satyanarayanan:
Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions
School of Computer Science, Carnegie Mellon University, Pittsburgh
CMU-CS-95-126, March 1995
<ftp://reports.adm.cs.cmu.edu/usr/anon/1995/CMU-CS-95-126.ps>

[32] Raquel **Menezes**, Carlos Baquero & Francisco Moura:
A Portable Lightweight Approach to NFS Replication
Proc. of ROSE'94 Conference, Bucharest, November 1994.

[33] Lily B. **Mummert**, Maria R. Ebling & M. Satyanarayanan:
Exploiting Weak Connectivity for Mobile File Access
School of Computer Science, Carnegie Mellon University
CMU-CS-95-185, 1995
<ftp://reports.adm.cs.cmu.edu/usr/anon/1995/CMU-CS-95-185.ps>

[34] Jeppe Damkjær **Nielsen**:
Transactions in Mobile Computing
Department of Computer Science, University of Copenhagen, Denmark
Written Work no. 95-2-11, spring 1995.
<ftp://ftp.diku.dk/diku/distlab/amigos/diku-95-2-11.ps.gz>.

[35] Brian D. **Noble** & M. Satyanarayanan:
An Empirical Study of a Highly Available File System
School of Computer Science, Carnegie Mellon University
CMU-CS-94-120, February 1994
<ftp://reports.adm.cs.cmu.edu/usr/anon/1994/CMU-CS-94-120.ps>

[36] **ORACLE**:
PL/SQL User's Guide and Reference
Version 1.0, April 1989 (Revised November, 1990)

[37] **ORACLE**:
RDBMS Database Administrator's Guide
Version 6.0, November 1988 (Revised October, 1990)

[38] **ORMC'96** discussions
Workshop on Object Replication and Mobile Computing
OOPLSA'96, San José, California, October 7, 1996

[39] Karin **Petersen** *et al.*:
Bayou: Replicated Database Services for World-wide Applications
<http://mosquitonet.stanford.edu/sigops96/papers/petersen.ps>

[40] Evaggelia **Pitoura** & Bharat Bhargava:
Maintaining Consistency of Data in Mobile Distributed Environments
Department of Computer Sciences, Purdue University, West Lafayette
15th Int. Conference on Distributed Computing Systems (ICDCS), 1995
Long version

[41] John **Saldanha**:
A File System for Mobile Computing
Dissertation Proposal
Department of Computer Science, University of Notre Dame, Indiana
Technical Report 93-17, December 1993

[42] M. **Satyanarayanan** *et al.*:
Experience with Disconnected Operation in a Mobile Computing Environment
School of Computer Science, Carnegie Mellon University
Proc. of the Mobile & Location-Independent Computing Symposium
USENIX Association, August 2-3, 1993

[43] M. **Satyanarayanan**, Brian Noble, Puneet Kumar & Morgan Price:
Application-Aware Adaptation for Mobile Computing
School of Computer Science, Carnegie Mellon University
CMU-CS-94-183, July 28, 1994
<ftp://reports.adm.cs.cmu.edu/usr/anon/1994/CMU-CS-94-183.ps.Z>

[44] M. **Satyanarayanan**:
Mobile Information Access
School of Computer Science, Carnegie Mellon University
CMU-CS-96-107, January 1996
IEEE Personal Communications 3(1), February 1996
<ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-107.ps>

[45] M. **Satyanarayanan**:
Fundamental Challenges in Mobile Computing
School of Computer Science, Carnegie Mellon University
CMU-CS-96-111, 1996
<ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-111.ps>

[46] Alex **Siegel**, Kenneth Birman & Keith Marzullo:
Deceit: A Flexible Distributed File System
Cornell University, Ithaca, NY
December 7, 1989

[47] Avi **Silberschatz** & Peter Galvin:
Operating System Concepts
4th edition, Addison-Wesley, 1994

[48] W. Richard **Stevens**:
UNIX Network Programming
Prentice Hall, 1990.

[49] **Sun** Software Technical Bulletin, February 1993:
"PC-NFS: Open Network Computing for PCs"
CD-ROM: "SunSolve 2.1, Sun Technical Bulletin: Dec 91-Oct 93", 1993
<http://www.diku.dk/software/sunos413-stb/1119.ps>.

[50] Michael **Svendsen**:
Eksempel client/server applikation
DistLab e-mail
Datalogisk Institut, Københavns Universitet.

[51] Michael G. **Sørensen**:
A Model for Multi-Level Consistency
OOPSLA'96 Workshop on Object Mobility and Replication (ORMC'96)

[52] Andrew S. **Tanenbaum**:
Distributed Operating Systems
Prentice-Hall, 1995.

[53] Douglas B. **Terry** *et al.*:
Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System
Computer Science Laboratory, Xerox Research Center, Palo Alto

[54] Douglas B. **Terry**
Personal discussion
Xerox PARC Research Center, October 14, 1996.

Index

> "Most books have indexes; most technical reports don't. They should.
> Any nonfiction work of more then twenty or so pages that is worth reading deserves an index."
> – Leslie Lamport (LaTeX2ϵ), 127